

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

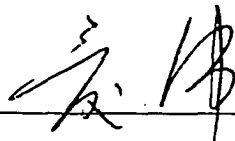
--

-

**A SYSTEM DYNAMICS MODEL FOR CONCURRENT
SOFTWARE ENGINEERING**

The members of the Committee approve the doctoral
dissertation of Chih-tung Hsu

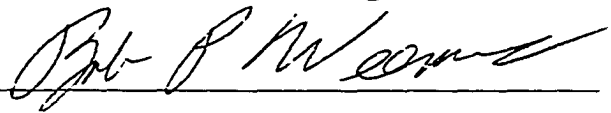
Pei Hsia
Supervising Professor



David C. Kung



Bob P. Weems



Lawrence B. Holder



Piotr J. Gmytrasiewicz



Dean of the Graduate School



Copyright © by Chih-tung Hsu 1999

All Rights Reserved

**A SYSTEM DYNAMICS MODEL FOR CONCURRENT
SOFTWARE ENGINEERING**

by
CHIH-TUNG HSU

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 1999

UMI Number: 9948006

Copyright 1999 by
Hsu, Chih-tung

All rights reserved.

UMI[®]

UMI Microform 9948006

Copyright 2000 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

ACKNOWLEDGMENTS

First of all, I would like to express my deep appreciation to my thesis supervisor, Dr. Pei Hsia, for his enduring guidance, encouragement, and support throughout the years. My special thanks to Dr. David Kung for his guidance and support in the past few years.

I would like to acknowledge Dr. Bob Weems, Dr. Lawrence Holder, and Dr. Piotr Gmytrasiewicz for serving on my Ph.D. committee and providing guidance. I took three courses from Dr. Weems, two courses from Dr. Holder, and have assisted Dr. Gmytrasiewicz in the past. Their teaching attitudes and abilities have motivated me to become a good teacher.

Special thanks to Professor Mikio Aoyama and his colleagues at Fujitsu, Kawasaki, Japan, for sharing with me their experience in concurrent software development projects. And special thanks to my best friend Pei-ching for reviewing and editing the thesis.

Finally, I would like to express my deep gratitude to my parents for their understanding and everlasting love and support and to my sisters, Coco and Abby, and their families, for their love and enduring support. I could not have done it without them.

May 12, 1999

ABSTRACT

A SYSTEM DYNAMICS MODEL FOR CONCURRENT SOFTWARE ENGINEERING

Publication No. _____

Chih-tung Hsu, Ph.D.

The University of Texas at Arlington, 1999

Supervising Professor: Pei Hsia

Concurrent engineering (CE) has been widely adopted and has made significant contributions to the electronics and manufacturing industries in terms of project cost and cycle time reduction, as well as product quality improvement. The software development industry has begun to learn from the CE experiences as practiced in other industries. Several software companies have significantly reduced their product cycle time by applying a modest degree of concurrent engineering; for example, Fujitsu's Concurrent Development model, Microsoft's Daily Build process, HP's Platform Development model, concurrent internationalization of software products for local markets, and DuPont's Timebox approach.

Concurrent software engineering (CSE) shortens time-to-market but creates new problems in terms of coordinating multiple, concurrent activities. The extent of benefits that CSE-based practices can deliver, their critical success factors, and the potential high risk areas need to be assessed carefully.

This research aims to develop a system dynamics simulation model (CSE-SD) to systematically assess the benefits and drawbacks of CSE. We made three major contributions in this research: (1) we have classified different types of CSE practices; (2) we have identified the specific benefits, potential risks, and the dynamic cause-effect implications of different types of CSE practices; and (3) we have studied three sets of questions using this system dynamics model. The results of our study provide strategic information for software project managers who attempt concurrent software development.

The CSE-SD model is an economic and effective management policy exploration tool for pre-assessing the benefits and potential risks of future projects. By calibrating the simulation model against the data collected from previous projects, it can be used to predict the possible outcomes of different management policies, actions, or decisions.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iv
ABSTRACT	v
LIST OF FIGURES	xii
LIST OF TABLES	xvii
Chapter	
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Objectives and Expected Significance	1
1.3 Research Approach	2
1.4 Organization of the Thesis	4
2. BACKGROUND	6
2.1 Introduction	6
2.2 Concurrent Engineering	6
2.2.1 Definition of Concurrent Engineering	6
2.2.1.1 Concurrency.....	8
2.2.1.2 Integration	10
2.2.1.3 Information Sharing.....	11
2.2.1.4 Quality Focus	12
2.2.2 CE-based Process Improvement	12
2.2.2.1 Cycle Time Reduction	13

2.2.2.2 Quality Improvement	14
2.2.2.3 Cost Reduction	15
2.3 Concurrent Software Engineering	16
2.3.1 Concurrency	16
2.3.2 Integration	17
2.3.3 Information Sharing	18
2.3.4 Quality Focus	18
2.3.5 Concurrent Software Engineering Framework	19
2.4 System Dynamics	21
3. CONCURRENT SOFTWARE ENGINEERING FRAMEWORK	24
3.1 Introduction	24
3.2 A RAW Model	24
3.3 A Classification of CSE Models	27
3.3.1 Type 1 Concurrency	28
3.3.2 Type 2 Concurrency	28
3.3.3 Type 3 Concurrency	30
3.3.4 Type 0 Concurrency	31
3.4 State-of-the-Practice CSE Practices	32
3.4.1 Concurrent Development Model	33
3.4.2 Concurrent Internationalization	34
3.4.3 Platform Development Model	36
3.4.4 Parallel Timebox Development	38
3.4.5 Hardware-Software Codesign	39

3.4.6	The IPTES Approach	41
3.4.7	Microsoft Daily Build Process	43
4.	A SYSTEM DYNAMICS MODEL	45
4.1	Introduction	45
4.2	Dynamics of Concurrent Software Engineering	46
4.2.1	Phase Overlapping	46
4.2.2	Synchronous Concurrent Subsystems	51
4.2.3	Asynchronous Concurrent Subsystems	55
4.2.4	Cross Function Integration	57
4.3	Model Structure	62
4.4	Comparison with Other Related SD Models	68
4.4.1	Abdel-Hamid and Madnick	68
4.4.2	JPL	69
4.4.3	Madachy	70
4.4.4	Collofello and Tvedt	71
5.	MODEL TESTING	75
5.1	Introduction	75
5.2	Unit Testing	76
5.3	System Testing	95
6.	BROOKS' LAW REVISITED	101
6.1	Introduction	101
6.2	Related Studies on Brooks' Law	101
6.3	Dynamics of Brooks' Law	103
6.4	Simulation Results	107

6.5 Summary	113
7. ON THE IMPACT OF CONCURRENT SOFTWARE ENGINEERING	115
7.1 Introduction	115
7.2 Model Calibration	116
7.2.1 The BASELNE Software Project	116
7.2.2 Mapping COCOMO Development Activities to CSE-SD	117
7.2.3 Calibrate CSE-SD Against COCOMO	122
7.3 Impact of Phase Overlapping	126
7.3.1 Modeling Phase Overlapping	127
7.3.2 Modeling Requirements Changes	128
7.3.3 Simulation Results	131
7.4 Impact of Synchronous Concurrent Subsystems	137
7.4.1 Determining Communication Overhead	138
7.4.2 Interteam Interactions	140
7.4.3 Experimentation Setting	143
7.4.4 Simulation Results	147
8. CONCLUSIONS AND FUTURE WORK	154
8.1 Contributions of the Research	154
8.2 Important Findings	155
8.2.1 Brooks' Law	156
8.2.2 Impact of Phase Overlapping	157
8.2.3 Impact of Synchronous Concurrent subsystems	157

8.3 Future Work	158
Appendix	
A. CSE-SD MODEL SPECIFICATION	160
B. CSE-SD MODEL EQUATIONS	201
C. KEY STATISTICS OF THE EXAMPLE PROJECT	231
REFERENCES	233
BIOGRAPHICAL INFORMATION	243

LIST OF FIGURES

Figure	Page
2.1. Traditional sequential engineering	9
2.2. Concurrent engineering	10
2.3. A fishbone diagram for the reasons of CE-based process improvement	15
2.4. The Blackburn CSE framework	20
3.1. A conceptual resource-activity-work product model	25
3.2. Type 1 concurrency	28
3.3. Type 2 concurrency	29
3.4. Type 3 concurrency	30
3.5. Type 0 concurrency	31
3.6. The Fujitsu concurrent development model	33
3.7. Concurrent internationalization of global software products	35
3.8. The platform development model	37
3.9. The parallel timebox development practice	39
3.10. Hardware-software codesign	41
3.11. The IPTES approach	42
3.12. The Microsoft daily build process	44
4.1. Dynamics of phase overlapping	47
4.2. Dynamics of synchronous concurrent subsystems	51
4.3. Dynamics of asynchronous concurrent subsystems	56
4.4. Dynamics of cross function integration	58
4.5. Overview of the CSE-SD model	67

5.1. Project progress of a perfect project	77
5.2. Adjusting the planned project effort when there is a reported gap between the perceived project effort needed to complete the project and the remaining project effort	79
5.3. Nominal and actual development defect rate	80
5.4. The impact of defect density on development defects generation	81
5.5. The impact of workforce mix on development defects generation	83
5.6. The impact of schedule pressure on development defects generation	85
5.7. Project scope change	87
5.8. Training time	88
5.9. Slack time and overtime	89
5.10. Learning effect on staff production rate	91
5.11. The impact of staff exhaustion level on staff production rate	93
5.12. The effect of schedule pressure on staff production rate	95
5.13. Comparison of project progress of the EXAMPLE project	98
5.14. Comparison of project cost of the EXAMPLE project	99
5.15. Comparison of scheduled completion date of the EXAMPLE project	99
5.16. Comparison of work force distribution of the EXAMPLE project	100
6.1. The dynamics of Brooks' Law	105
6.2. Modeling sequential constraint	107

6.3.	The impact of work force stability on project duration and cost	108
6.4.	The impact of degree of concurrency on project duration and cost	110
6.5.	Impact of restaffing time on project duration, cost, and number of needed work force	112
7.1.	Planned work force distribution	123
7.2.	Staffing plan stability	124
7.3.	Comparison of FTE software personnel distribution	124
7.4.	Comparison of cumulative project effort	126
7.5.	Modeling phase overlapping	128
7.6.	Rework cost ratio	130
7.7.	Three patterns of requirements change	130
7.8.	Project duration increase due to requirements changes	133
7.9.	Project effort increase due to requirements changes	134
7.10.	The effects of phase overlapping on project effort and development cycle time	136
7.11.	Determining the overall communication overhead	139
7.12.	Intrateam and interteam communication overheads	140
7.13.	Interteam interference amplification	142
7.14.	Interteam-to-intrateam communication ratio	144
7.15.	Project size change due to resolution of interteam interferences	145
7.16.	Project duration vs. number of teams (low communication ratio M1)	151
7.17.	Project effort vs. number of teams (low communication ratio M1)	151
7.18.	Project duration vs. number of teams (medium communication ratio M2)	152

7.19. Project effort vs. number of teams (medium communication ratio M2)	152
7.20. Project duration vs. number of teams (high communication ratio M3)	153
7.21. Project effort vs. number of teams (high communication ratio M3)	153

LIST OF TABLES

Table	Page
4.1. Major components of the CSE-SD model	62
7.1. Phase distribution of project effort, schedule and personnel	117
7.2. The breakdown of project effort, schedule, and personnel in the "Plan and Requirements" phase	119
7.3. The breakdown of project effort, schedule, and personnel in the "Product Design" phase	119
7.4. The breakdown of project effort, schedule, and personnel in the "Programming" phase	120
7.5. The breakdown of project effort, schedule, and personnel in the "Integration and Test" phase	120
7.6. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 0%	120
7.7. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 10%	121
7.8. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 20%	121
7.9. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 25%	121
7.10. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 30%	122

7.11. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 40%	122
7.12. Comparison of project effort (person-days)	125
7.13. Nominal project (R1xD1) with different requirements change patterns	134
7.14. Modest phase overlapping (R2xD2) with different requirements change patterns	135
7.15. Aggressive phase overlapping (R3xD2) with different requirements change patterns	135

CHAPTER 1

INTRODUCTION

1.1 Motivation

Concurrent engineering (CE) has been widely adopted and has made significant contributions to the electronics and manufacturing industries in terms of project cost and cycle time reduction, as well as product quality improvement. The software development industry has begun to learn from the CE experiences as practiced in other industries. Several software companies have significantly reduced their project development cycle time by applying a modest degree of CE; for example, Fujitsu's Concurrent Development model [12-19], Microsoft's Daily Build process [34-35], HP's Platform Development model [44], concurrent internationalization of software product for local markets [65], and DuPont's Timebox approach [54].

Concurrent software engineering (CSE) shortens time-to-market but creates new problems in terms of coordinating multiple, concurrent activities. The extent of benefits that CSE-based practices can deliver, their critical success factors, and the potential high risk areas need to be assessed carefully.

1.2 Objectives and Expected Significance

The overall objectives of this research are: (1) to classify the unconventional software development paradigms according to their concurrent software engineering characteristics; and (2) to construct a system dynamics model for

assessing the benefits, and drawbacks, if any, of concurrent software engineering in terms of project cost and development cycle time reduction.

Concurrent engineering (CE) principles have been adopted widely and with great success in the manufacturing industry. Although some CE principles are being cautiously adopted by software producers, the potential benefits of CE may not be fully realized in the software industry. We expect that this research will construct a comprehensive system dynamics model (called CSE-SD) that allows us to systematically assess the benefits and potential risks in adopting CE principles in software development. It will advance the state of the art and practice of concurrent software engineering and substantially improve current software development practices in terms of project cost and development cycle time reduction.

By calibrating the system dynamics simulation model against the data collected from previous projects, the proposed model can be used as a management policy exploration tool for future projects. The model can help project managers predict the possible outcomes of different management policies, actions, or decisions. The proposed concurrent software engineering system dynamics (CSE-SD) model can be employed to answer questions such as: “What is the impact of concurrent development on project cost and development cycle time?”; “Will concurrent development reduce project cost and development cycle time?”; “Under what situations will concurrent development have the most leverage?”; “How many concurrent development teams are suitable for the project?”; and, “What is the optimal degree of concurrency in terms of project duration and cost?”

1.3 Research Approach

In software engineering, it is remarkably easy to propose hypotheses and remarkably difficult to test them. Accordingly, it is useful to seek methods for testing

software engineering hypotheses [81]. Unfortunately, conducting experiments in the area of software development is costly and time-consuming [58]. Conducting experiments in software is difficult and problematic for several reasons. First, software development is a complex process, involving numerous factors which do not remain constant throughout the period of experimentation. It is difficult to control one factor while keeping all other factors constant. Second, while the results derived from an experiment might be meaningful and useful to a specific environment and context, it is not generally applicable to other environments and contexts. Third, controlled experimentation is not feasible for large-scale projects due to the exponential growth in the number of factor combinations as the number of factors under study increases. Studying the impact of a new software development methodology and/or process on schedule, quality, and cost in the development of a large-scale system is infeasible, although not impossible. Finally, participating engineers generally have to spend extra time reporting measurements, which takes away from the time they spend on productive work.

In this research, we use the System Dynamics (SD) simulation approach to study the impact of concurrent software engineering on project cost and development cycle time. System Dynamics refers to a quantitative method to investigate the dynamic behavior of socio-technical systems and their responses to policy [77]. It was developed by Jay Forrester in 1961 and, since then, has been applied to many different fields. A review of the approach and its application in software project management is presented in chapter 2.

Simulation models, like empirical cost-estimation models, can be used to predict the schedule and cost for future projects to be developed, once the models are calibrated against specific development environments and organizations. In a simulation-based experiment, the effect of changing one factor can be observed while all

other factors are held unchanged. Software project managers can easily assess the impact of different development strategies and policies simply by changing the values of individual model parameters [7].

The proposed system dynamics model is calibrated according to three different sources: (1) industrial or experimental data published in the literature; (2) interviews with project managers in Fujitsu, and (3) data derived from the COCOMO cost estimation model.

The proposed system dynamics model CSE-SD can be used: (1) to simulate the proposed development process and various software project management policies; (2) to test the impact of various assumptions, scenarios, and environmental factors on the software development process; (3) to predict the consequences of management actions on the interrelationships among software development process components and flows, and (4) to examine the sensitivity of the software development process to various internal and external factors [48].

1.4 Organization of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents fundamental principles of concurrent engineering as practiced in the manufacturing industry and identifies main reasons why they improve the hardware development process. Related work on the system dynamics approach and concurrent software engineering practices are briefly reviewed in this chapter.

Chapter 3 presents a systematic classification of various concurrent software engineering practices based on a proposed resource-activity-work product (RAW) model. A detailed review of the state-of-practice concurrent software engineering practices based on the RAW is presented.

Chapter 4 presents a system dynamics model for evaluating the impact of concurrent software engineering practices. The benefits, potential risks, and critical factors, as well as the dynamic cause-effect interrelationships of each type of CSE, are discussed. These dynamic cause-effect relationships serve as the basis from which the proposed system dynamics model is developed. Finally, four related software project system dynamics models are reviewed and compared.

The results of model testing are presented in chapter 5. Model testing is performed in two steps: unit-level testing and system-level testing. Unit-level testing concerns the correctness of individual model sectors, while system-level testing integrates and tests all model components. The model-simulated behaviors are compared with those of the Abdel-Hamid and Madnick model [7].

In chapters 6 and 7, we conduct a set of simulation experiments to further demonstrate the capability of CSE-SD in generating useful information and insights for software project managers. Chapter 6 addresses the issue of project restaffing, and testing the validity of Brooks' Law. Specific questions addressed in chapter 7 include: (1) the impact of the *phase overlapping* concurrent development approach on project cost and development cycle time; and (2) the impact of the *synchronous concurrent subsystems* development approach on project cost and development cycle time.

The results of this research are concluded and summarized in chapter 8. A number of questions and issues that merit further study are also discussed. A detailed specification of the CSE-SD model, including formal model equations, is given in appendices A and B.

CHAPTER 2

BACKGROUND

2.1 Introduction

In this chapter we present fundamental principles of concurrent engineering as practiced in the manufacturing industry and identify the main reasons why they improve the hardware development process. Related work on the system dynamics approach and concurrent software engineering practices and framework are briefly reviewed in this chapter. A more detailed presentation of the state-of-practice concurrent software engineering practices based on a proposed resource-activity-work product (RAW) model is included in chapter 3. Related software project system dynamics models are compared with the proposed CSE-SD model in chapter 4.

2.2 Concurrent Engineering

In this section, we review and define concurrent engineering and examine the reasons for CE-based process improvements.

2.2.1 Definition of Concurrent Engineering

Since it became a recognized technique in the mid-1980s, concurrent engineering (CE) has made significant contributions to the electronics and manufacturing industries in terms of project cost and cycle time reduction, as well as product quality improvement. Unfortunately, there is no well-accepted definition of CE. Some researchers describe CE as a parallel design approach, while others emphasize the cross-functional design team approach. For others, CE simply refers to a group of

sound principles, contemporary techniques and novel methodologies that help improve the product development process. Some of the most-cited definitions of CE are:

1. Concurrent engineering is a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. This approach is intended to cause the developers, from the outset, to consider all elements of the product life-cycle, from conception through disposal, including quality, cost, schedule, and user requirements [85].
2. The Computer-aided Acquisition and Logistics Support program (CALS) definition of CE is “a systematic approach to creating a product design that considers all elements of the product life cycle, from conception to disposal. CE defines simultaneously the product, its manufacturing process, and all other required life-cycle processes, such as logistic support. CE is not the arbitrary elimination of a phase of the existing, sequential, feed-forward engineering process, but rather the co-design of all downstream processes toward a more all-encompassing, cost-effective optimum. Concurrent engineering is an integrated design approach that takes into account all desired downstream characteristics during upstream phases to produce a more robust design that is tolerant of manufacturing and use variation, at less cost than sequential design [27].
3. CE is a goal-directed effort, where “ownership” is assigned mutually among the entire group on the “total job” to be completed, not just a “piece” of it, with the understanding that the team is empowered to make major design decisions along the way [78].
4. CE is a product development methodology where up-front “X-abilities” (such as manufacturability, serviceability, and quality) are considered part of the product design and development process. X-abilities are not merely for meeting the basic

functionality or a set of limited strategies, but for defining a product that meets all the customer requirements [69].

5. Concurrent engineering is a term that has been applied since the 1980s to the product development process where, typically, a product design and its manufacturing process are developed simultaneously, cross-functional groups are used to accomplish integration, and the voice of the customer is included in the product development process [76].

The above frequently cited definitions of CE and others ([41], [60]) spell out four key characteristics of concurrent engineering: concurrency, integration, information sharing, and quality focus.

2.2.1.1 Concurrency

The trademark characteristic of CE is activity concurrency. In traditional product development projects, each stage of the project is done sequentially, with the functional groups “handing-off” the project to one another after an extensive stage-gate evaluation process [85]. A generic traditional sequential engineering (SE) process is shown in figure 2.1. In SE, the product design group, upon receipt of a complete product specification from the marketing department, performs product design in an environment isolated from all other departments. Only after a design is verified, either by simulation or hardware prototyping or both, is it handed off to manufacturing, test, quality, and service engineers for review [67].

Design flaws and test failures detected during manufacturing are reported back to the product design department for diagnostics. The product design group reworks the design and “tosses it over the wall” to the manufacturing department. This redo-until-right practice, involving many toss-it-over-the-wall rework iterations, usually is a lengthy and costly process.

CE replaces SE with simultaneous performance of activities. Concept development, product design, and process design are performed at the same time. As shown in figure 2.2, all downstream issues such as manufacturability, quality, serviceability, product performance, cost, and other downstream X-abilities are considered early in the product design stage. The “do-it-right-the-first-time” philosophy of CE replaces the lengthy “redo-until-right” philosophy as practiced in SE.

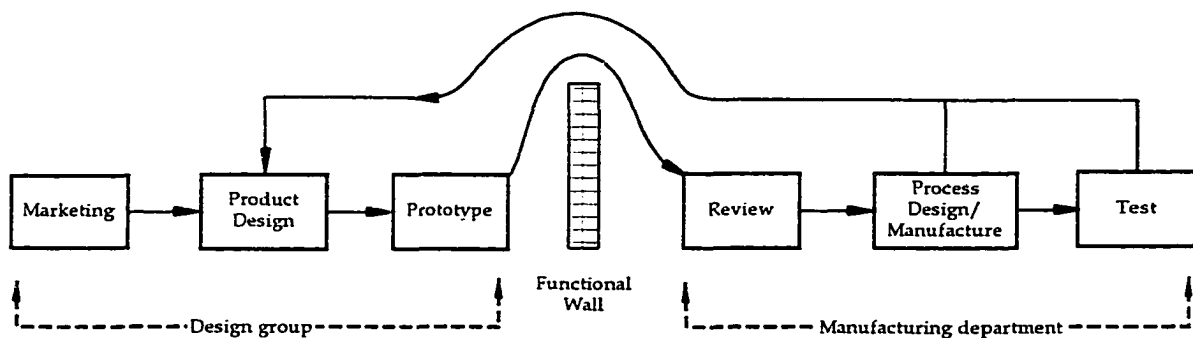


Figure 2.1. Traditional sequential engineering.

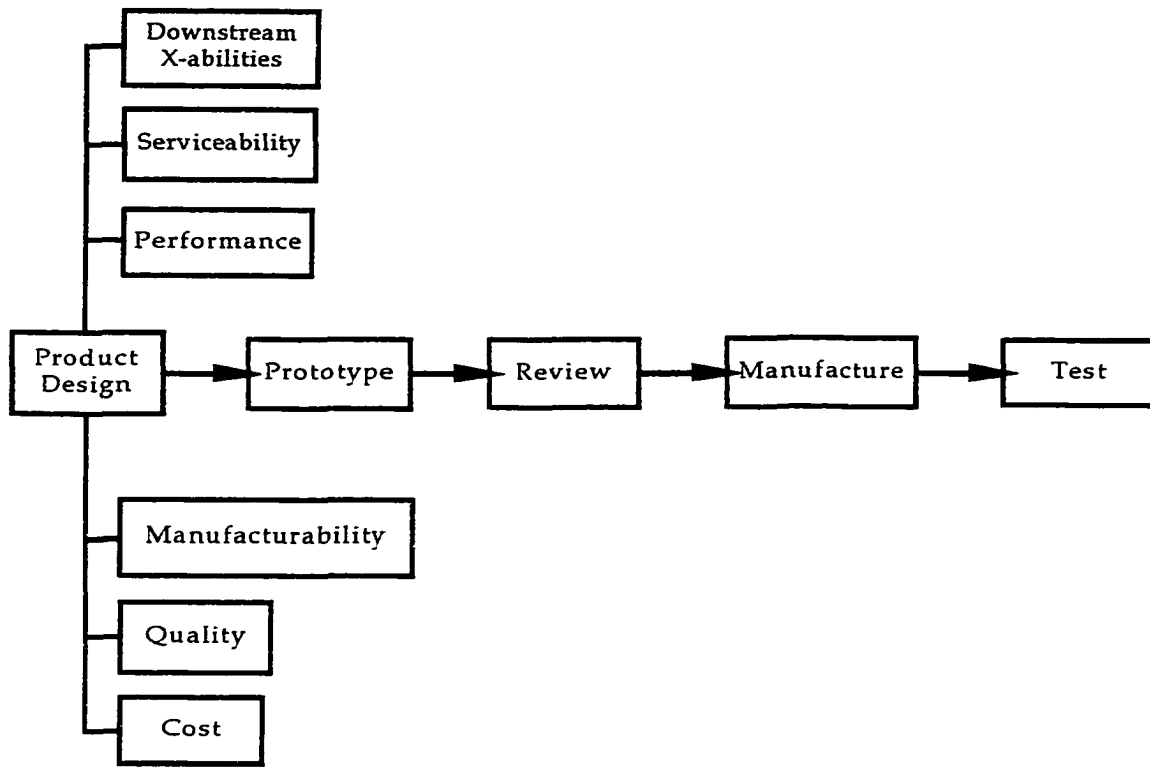


Figure 2.2. Concurrent engineering.

2.2.1.2 Integration

The second key characteristic of CE is integration: integration of design and manufacturing (design-manufacturing integration) and integration of customer and design (marketing-design integration) [76]. Integration refers to the up-front involvement of personnel from different functional areas, including marketing, product design, process design, manufacturing, service, or other relevant areas, depending on the type of product.

Usually, the mechanism for accomplishing integration is the use of cross-functional teams. People from many departments collaborate over the life of a product—from idea to obsolescence—to ensure that it reflects customers' needs and desires [67].

Team members usually stay together physically. They are empowered to make timely decisions in shaping the design, and are given ownership of what they produce, and are rewarded as a whole on a team basis.

2.2.1.3 Information sharing

Information sharing is another key characteristic of CE. Three forms of information sharing take place in CE-based projects: flying start, front loading, and two-way information exchange [21]. These three forms of information sharing are identified based on the direction of information flows between development phases.

Flying start is a preliminary information transfer flowing from upstream design activities to team members primarily concerned with downstream activities. Early release of preliminary information supports CE by enabling downstream activities to start earlier.

Front loading is the early involvement in product design activities of downstream issues such as manufacturing, testing, and service. Design techniques and practices, such as design for manufacturability and assembly (DFM/DFA), design for testability (DFT), and other design for X-abilities, specify ways and suggest rules to design products that are easy to manufacture and test. For example, “reducing the number of parts,” “simplifying the part mating and securing processes,” and “creating symmetry or asymmetry so that it is difficult to put the parts together in any manner but the correct way” are common DFA design rules that consider downstream assembly processes early in the product design phase.

Two-way information exchange is intensive and rich communication between teams while performing concurrent activities [21]. Teams involved in concurrent development of different subsystems (e.g., hardware and software) need to have a

steady flow of information among the groups to prevent potential integration problems [21].

2.2.1.4 Quality Focus

The fourth key characteristic of CE is quality focus, both on the product and on the process that produces it. CE not only is concerned with quality control of the product, but it also focuses on continuously improving the process itself [74]. Many techniques, such as total quality management (TQM), quality function deployment (QFD), just-in-time manufacturing (JIT), statistical process control (SPC), and so forth, are employed to ensure that quality standards and objectives are met. TQM applies a set of principles to focus continuous attention on quality at every step of design, development, and manufacturing [67]. QFD methods are designed to listen to the voices of customers [67]. A set of matrices relating subjective customer desires to quantitative engineering characteristics is employed to address the needs of the customer throughout the entire product development process.

CE seeks ways to continuously improve product quality and process effectiveness. Rather than try to find defects in finished products, statistical process control (SPC) seeks to monitor and correct drifts in quality in the manufacturing process [67]. CE continuously seeks ways to improve the development process (continuous process improvement).

2.2.2 CE-based Process Improvement

The goal of concurrent engineering is to cut project cost and development cycle time and improve product quality, all at the same time. In this section, we examine the underlying reasons behind CE-based process improvements.

2.2.2.1 Cycle time reduction

CE reduces product development cycle time (refer to Bone 1 of figure 2.3) mainly because of three reasons: concurrent activities (Bone 2), less rework (Bone 3), and reacting to changes quickly (Bone 4). Activity concurrency is the major force of concurrent engineering. Concurrency of activities has contributed significantly to the cycle time reduction in the manufacturing industry ([67], [74]). The overall product development cycle time is reduced because the steps along the way are handled in parallel instead of series, as usual [67].

Rework is reduced mainly because of two reasons: shorter rework loop (Bone 5) and fewer requirements and design changes (Bone 6). CE shortens the rework loop both by shortening the interval between the time defects are introduced and the time they are detected (defect-to-correct distance, shown as Bone 7) and reducing the number of rework iterations (Bone 8). Because of cross-function integration, problems are identified early. Rework does not need to go through the lengthy toss-it-over-the-wall iterations between design and manufacturing.

Requirements and design changes are fewer because of cross-function integration: design-manufacturing integration and design-marketing integration. Design-manufacturing integration allows downstream issues to be considered early (Bone 9) in the product design stage (i.e., front loading information sharing). This leads to early problem identification, and a more robust and manufacturable design. Design changes are reduced when product development goes to the manufacturing stage.

CE focuses on the needs of the customer early and throughout the entire development process (Bone 10). Early and continuous involvements of customers (design-marketing integration, or design-customer integration) help the designer and the customer negotiate the requirements and arrive at a stable product specification

early. Usually, QFD methods (Bone 11) are designed to make sure the voice of the customer is included in the product development process [76].

Another reason that CE reduces product development cycle time is its capability to react quickly to changes (Bone 4). CE responds to changes quickly because of the empowerment of decision-making authority (Bone 12) and real-time communication among team members (Bone 13). Empowerment of decision-making authority allows team members to make timely decisions without waiting for long, upper-management approvals.

Real-time communication is facilitated by co-located cross-functional teams (Bone 14). In a cross-functional team setting (Bone 15), team members can discuss different strategies to implement the project and resolve problems together, instead of communicating across isolated functional groups. Locating team members close together also facilitates communication.

2.2.2.2 Quality Improvement

CE improves product quality (Bone 16) because of two main reasons: customer focus (Bone 17) and continuous process improvement (Bone 18). Quality, as defined by the customer, is improved because of early and continual customer focus throughout the entire development process. Customer satisfaction is maximized because their voice is echoed in every step of the development process.

Quality products come from quality processes. To produce a quality product that maximizes customer satisfaction and minimize negative product defects, CE seeks ways to improve the process and focus continuous attention on quality at every step of design, development, and manufacturing.

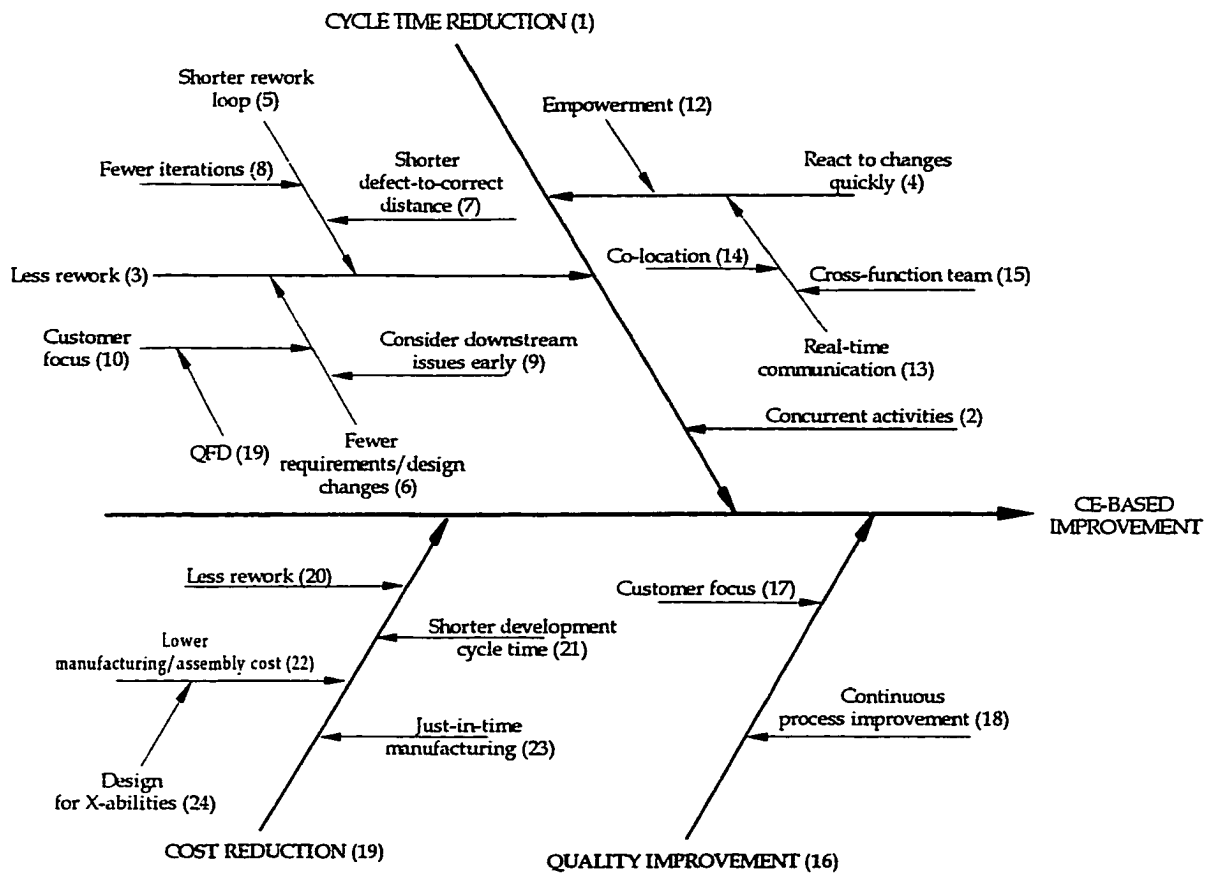


Figure 2.3. A fish bone diagram for the reasons of CE-based process improvement.

2.2.2.3 Cost Reduction

CE cuts project cost (Bone 19) mainly because of four reasons: less rework (Bone 20), shorter development cycle time (Bone 21), lower downstream manufacturing and assembly cost (Bone 22), and just-in-time manufacturing (Bone 23).

Less rework not only contributes to the reduction of development cycle time, but also helps to cut development cost. Design flaws are detected early in the product design stage, and they are corrected with less cost. By focusing on the needs of

the customer early, requirements changes and defects are reduced. With the reduced amount of rework tasks because of fewer changes and design flaws, project cost also is reduced.

The philosophy of “Design for X-abilities” (Bone 24) helps to produce a product design that is easy to manufacture, assemble, and test. For example, the DFA rules such as “reducing the number of parts” and “simplifying the part mating and securing processes” help to simplify the assembly process. This simplification has the effect of reducing direct assembly costs, and often tends to reduce indirect costs such as incoming inspection and parts inventories [76].

Just-in-time (JIT) manufacturing methods provide components and assemblies as they are needed. These components and assemblies make it unnecessary to maintain large inventories, and thus help to cut costs [67].

2.3 Concurrent Software Engineering

Despite its well-known problems, the sequential Waterfall model still is the software development process model most commonly used. In this section, we review literature that reports successes in applying CE principles to the software engineering community. We define concurrent software engineering (CSE) as a development process and management practice that (1) helps to cut project cost and development cycle time, and improve product quality; and (2) possesses the four key characteristics of CE: concurrency, integration, information sharing, and quality focus.

2.3.1 Concurrency

Driven by the increasing pressure to bring new products to market faster, many software companies have practiced concurrent engineering. Examples include:

Fujitsu's Concurrent Development Model [12-19]; Microsoft's Daily Build process [34-35]; HP's Platform Development Model [44]; concurrent internationalization of global software development [65]; concurrent development of real-time systems [62-63]; and Parallel Timeboxes to the development of information systems [54].

Phase overlapping, as practiced in the hardware development industry, is not commonly practiced in software industry due to the unstable front-end of the software development life-cycle.

2.3.2 Integration

The "walls" among different functional areas (e.g., design and manufacturing) are taller in CE than in CSE. In CE, for example, designers (white collars) and manufacturers (blue collars) usually speak different domain languages, and have different thinking and backgrounds. Software development is a more creative endeavor; therefore, most of the participants of a software development project are white collars. The differences among the different experts (e.g., requirements analysts, designers, programmers) are less significant than those in CE. For example, a programmer may do some domain analysis and analysts may do some programming (prototyping is an example).

Incorporating expertise from different disciplines in CSE is easier than in CE. A cross-functional teaming approach has been practiced in the software development industry. For example, Microsoft's "feature teams" practice has contributed to the successful development and delivery of Visual C++ [55]. AT&T's "application development teams" approach has helped the company make on-time deliveries of multiple releases of a telecommunication software system [73]. Xerox's "chunking teams" practice has contributed to the successful development of the *Inconcert* workflow management system [1].

2.3.3 Information sharing

The “front loading” type of information sharing currently is being practiced in the software development domain, as well. Examples include “design for testability” ([46], [51], [82]) and “design for reusability” [66]. The objectives of design for testability are to reduce the cost and complexity of tests. Early consideration and estimation of testability in the design phase helps designers identify parts of the specification that are hard to test; then appropriate transformations can be proposed to enhance testability of the end product. Designing for large-scale reuse addresses the need for higher productivity in domain-specific application domains or product families.

The “two-way information exchange” type of information sharing has been practiced in software development, in particular, in firmware development. Teams or individuals involved in concurrent development of the hardware and the software component need to have a steady flow of information between them to prevent potential integration problems. Use of the “flying start” type of information sharing to support overlapping software development, however, is not a common practice in software development.

2.3.4 Quality Focus

Global competitiveness has forced many companies to view quality improvement as a vital task [40]. Like CE, the software development industry has begun to apply quality-oriented techniques to improve the quality of both the product and the process. Specifically, software development organizations endeavoring to improve the quality of software systems (by improving the quality of the software development process) recently have adapted QFD for the development of software ([25], [32], [39-40], [70], [71-72], [86]), especially during the requirements analysis phase.

Implementing QFD techniques to the front-end of the software development life cycle can lead to effective communication with users, fewer design changes, and increased analyst and programmer productivity [40].

In summary, the software development industry has recognized the potential benefits of concurrent engineering and cautiously applied it from different aspects and for different purposes. Phase overlapping and the “flying start” type of information sharing, however, are not common practices in software development, because of the unstable front-end.

2.3.5 Concurrent Software Engineering Framework

Blackburn et al. [21] proposed a framework for concurrent software engineering based on Clark and Fujimoto’s information processing framework for supporting overlapping problem solving activities [28]. The Blackburn framework, as shown in figure 2.4, distinguishes four types of activity concurrency (within-stage overlap, across-stage overlap, hardware/software overlap, and across-project overlap) and three forms of information concurrency (front loading, flying start, and two-way high bandwidth information exchange) [21].

Situated between activity concurrency and information concurrency are practices of “architectural modularity” and “synchronicity.” Architectural modularity, a critical issue for “within-stage overlap” and “across-project overlap,” is supported by front loading. Front loading (information about possible design changes, customer requirements, and reuse concerns) helps developers design more robust and modular system architectures with reusable modules.

Synchronicity is identified as a critical issue for the other two forms of activity concurrency: “across-stage overlap” and “hardware/software overlap.” Overlapping development and firmware development increase the degree of

coupling between overlapped phases and between hardware and software designers. Their work must be coordinated and synchronized to avoid late integration problems.

Poor problem decomposition and module designs increase the need for synchronizing concurrent activities within-stage (indicated as dotted line from synchronicity to within-stage overlap). Synchronicity is supported by all three types of information concurrency.

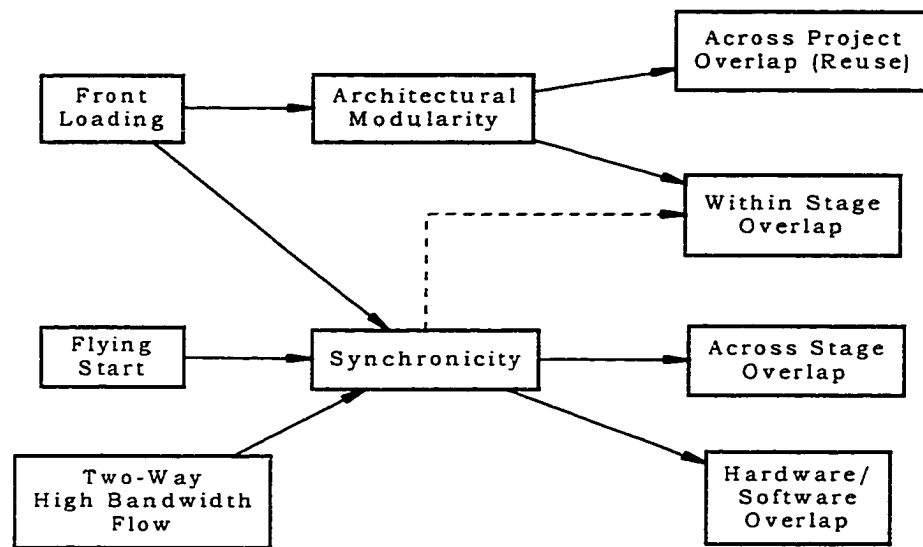


Figure 2.4. The Blackburn CSE framework.

The Blackburn CSE framework provides a coherent framework for CSE to move from *ad hoc*, reactive practices to proactive project management. However, the Blackburn framework is not appropriate to serve as a reference framework for our proposed system dynamics study, for two reasons. First, it considers only activity and information flow between activities; and other important issues, such as human resource and workload assignment, are not addressed. Second, it focuses on

development cycle time instead of the quality improvement and cost reduction potential of CE.

To guide the construction of the CSE system dynamics model, a more general and comprehensive framework is needed. In chapter 3 we classify different types of concurrency based on a proposed resource-activity-work product model. The benefits and potential risks of each type of concurrency are presented in chapter 4. Then, based on the classification and cause-effect analysis of CSE, a comprehensive system dynamics simulation model is constructed to quantitatively assess CSE.

2.4 System Dynamics

System Dynamics (SD) refers to a quantitative method to investigate the dynamic behavior of socio-technical systems and their responses to policy [77]. The field of system dynamics was developed initially from the work of Professor Jay W. Forrester in 1961 [36] as *Industrial Dynamics* and is defined as follows:

The study of the information-feedback characteristics of industrial activity to show how organizational structure, amplification (in policies), and time delays (in decisions and actions) interact to influence the success of the enterprise [36].

Since then, the application of SD has grown extensively and now encompasses numerous fields such as economics and finance, biology and medicine, corporate planning and policy design, transportation, banking, politics, energy and environment, and inflation and unemployment.

The fundamental philosophy of system dynamics is based on the premise that the behavior (or time history) of a system is caused principally by its underlying structure [9]. The general idea of SD can be described as consisting of three major steps: (1) eliciting important objects and variables, both tangible and intangible, that are believed to be responsible for generating the observed behavior; (2) identifying

their cause-effect relationships; and (3) constructing a quantitative model that encompasses and links all cause-effect feedback loops and analyzes the system as a whole. SD takes advantage of the fact that a computer model can be of much greater complexity and carry out more simultaneous calculations than can the mental model of the human mind.

Recently, the System Dynamics modeling technique has been applied to the software project management domain as well. The work of Abdel-Hamid and Madnick ([2], [7]) represents one of the first applications of SD in this area. Other works include Lin and Levary [48], Cooper [30-31], Madachy [52-53], Collofello and Tvedt [79-80], Rodrigues and Williams [68].

Cooper applied System Dynamics to software development projects with a focus on assessing the impacts of “work quality” and “rework discovery time” based on the generic concept of the rework cycle [30-31]. Their findings suggest that lowering rework discovery time is most leveraged in improving project schedule performance when quality is not at extremely low or extremely high levels [30]. Under low-quality conditions, software project managers should work first on quality improvement practices and systems such as early specification and design reviews, then accelerate rework discovery.

Rodrigues and Williams [68] proposed to integrate System Dynamics with traditional project management techniques to support the management of on-going projects. This is different from the conventional use of the system dynamics technique in which SD models are calibrated against completed projects and the diagnosis results from SD models are used to provide guidance for future project developments. In their work, the SD model is employed to assess the current plan, identify potential risks, diagnose segments of past behavior, and help identify causes

for deviations. The SD model is recalibrated to reproduce segments of past project behavior and provide new estimates for future behavior.

In summary, the above works represent important contributions for the application of SD to software project management. Whether it provides on-going dynamic support for the current project or postmortem analysis to provide guidance for future projects, SD has been found to offer important benefits to the analysis of software development project management. We will review the other four SD models (Abdel-Hamid and Madnick, Lin and Levary, Collofello and Tvedt, and Madachy) in more detail and compare them with the proposed CSE-SD model in section 4.4.

CHAPTER 3

CONCURRENT SOFTWARE ENGINEERING FRAMEWORK

3.1 Introduction

In this chapter we present a systematic classification of different types of CSE practices based on a conceptual Resource-Activity-Work product (RAW) model. The proposed RAW model provides the basis for us to define “concurrency,” identify different relationships that exist among resources, processes, and products, and, most importantly, to classify different types of CSE practices. In chapter 4, we will identify the benefits and potential risks of each type of CSE practice, then construct a system dynamics simulation model to quantitatively assess their impact.

The remainder of this chapter is organized as follows. Section 3.2 presents the proposed RAW model based on three entities: human resource, development activity, and work product. A classification of different types of CSE practices based on the RAW model is presented in section 3.3. We review and present state-of-the-practice CSE practices using the RAW model in section 3.4.

3.2 A RAW Model

Three entities are of concern: process, product, and resource [26]. These entities represent essential perspectives that most of the software process models need to capture. Processes are collections of all activities that are required to design and implement the software product. Requirements analysis, high-level design, detailed design, coding, test planning, and system integration and testing are common activities for any nontrivial software development projects. Products are any artifacts that are produced by

processes. For example, the “requirements specification” document is the work product produced by the “requirements analysis” activity. Resources are any items used by processes, excluding products of other processes. Human resources, for example, are the most important resource for any software development project.

We present a conceptual Resource-Activity-Work product (RAW) model to capture these three essential perspectives. The model considers the three entities at the same time and treats them as one integrated object.

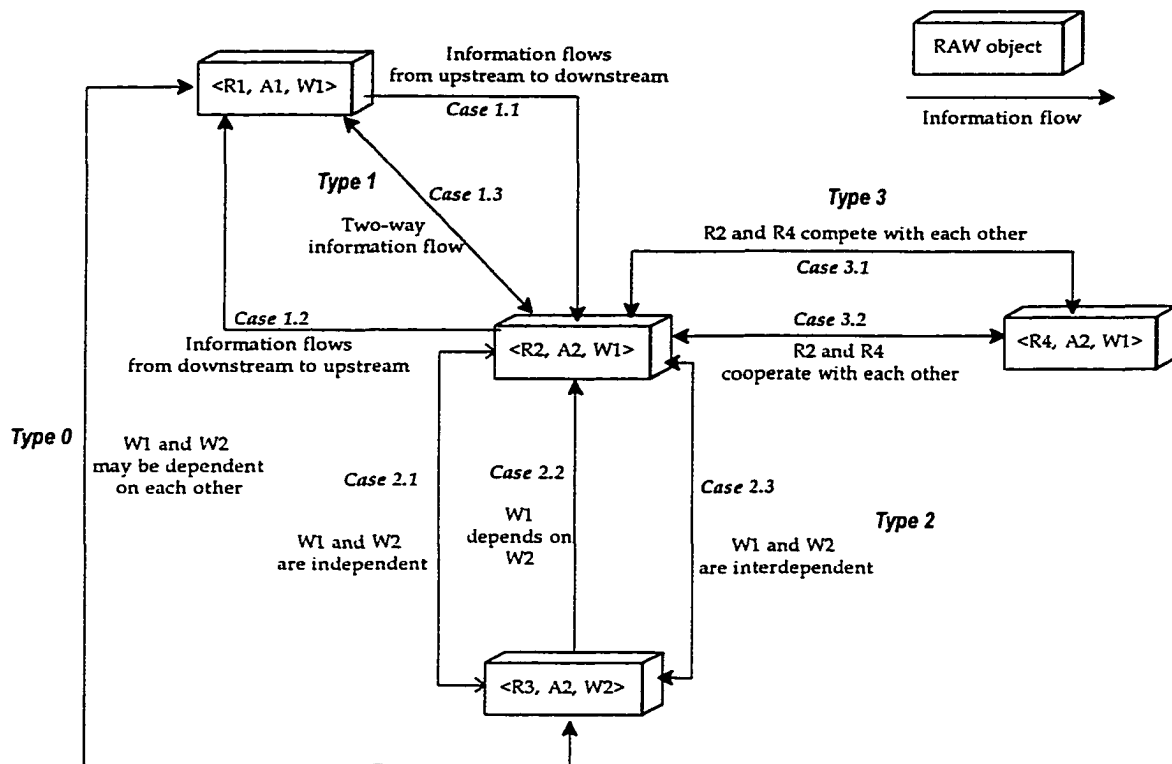


Figure 3.1. A conceptual resource-activity-work product model.

By treating human resources, development activities, and work products as one integrated object, we identify four major types of relationship between any two RAW objects, from Type 0 to Type 3, as illustrated in figure 3.1.

The “Type 0” relationship (e.g., between $\langle R1, A1, W1 \rangle$ and $\langle R3, A2, W2 \rangle$) captures the situation in which different teams or individuals (R1 and R3) perform different activities (A1 and A2) on different work products (W1 and W2). However, W1 and W2 may depend on each other. Here, the “R” component of the RAW object is coded as 0 (because of different human resources R1 and R3), the “A” component is coded as 0 (because of different activities A1 and A2), and the “W” is coded as 0 (because of different work products). Note that “0” means “different” while “1” represents “the same.”

The “Type 1” relationship (e.g., between $\langle R1, A1, W1 \rangle$ and $\langle R2, A2, W1 \rangle$) captures the situation in which different teams or individuals (R1 and R2) perform different activities (A1 and A2) on the same work product (W1). The Type 1 relationship can be further classified into three cases:

- Case 1.1: A2 depends on A1. An instance of this inter-RAW relationship is the traditional waterfall model, where information flows from the upstream phases to downstream phases. For example, design teams pass design specification to coding and testing teams for implementation and testing.
- Case 1.2: A1 depends on A2. Information flows from downstream phases to upstream phases. An example of this situation is “design for testability,” where testing issues are considered in the design phase.
- Case 1.3: A1 and A2 are interdependent. Information flows are bidirectional.

The “Type 2” relationship (e.g., between $\langle R2, A2, W1 \rangle$ and $\langle R3, A2, W2 \rangle$) captures the situation in which different teams or individuals (R2 and R3) perform the same activity (A2) on different work products (W1 and W2). Type 2 can be further classified into three different cases:

- Case 2.1: W1 and W2 are independent.

- Case 2.2: W1 depends on W2 (or W2 depends on W1). As an example, W2 is a program module that calls another module W1 (i.e., W2 depends on W1). Any changes to the interface of module W1 causes module W2 to be reworked, since it is affected.
- Case 2.3: W1 and W2 are interdependent.

The “Type 3” relationship (e.g., between $\langle R2, A2, W1 \rangle$ and $\langle R4, A2, W1 \rangle$) captures the situation in which different teams or individuals (R2 and R4) perform the same activity (A2) on the same work product (W1). The “Type 3” relationship can be divided into two cases, depending on how the two human resources are related to each other.

- Case 3.1: R2 and R4 compete with each other. They usually have the same skills or belong to the same functional groups. For example, two different programmers work on a shared program module. Concurrent updates to a common module may violate the integrity of that module.
- Case 3.2: R2 and R4 cooperate with each other. They usually have different skills or are members of different function groups. An example of this case is when members of a cross-functional team work on product design. For example, marketing specialists work with designers in drafting requirements specification for a new product.

3.3 A Classification of CSE Models

In this section, we classify CSE practices into different types based on the RAW model. We review state-of-the-practice CSE practices and demonstrate how they can be represented by the RAW model. To classify concurrent software development practices, we extend the RAW model with another dimension-time.

Definition. Each RAW object has a start time T_s and a finish time T_f . A RAW object is said to be active during the interval of $[T_s, T_f]$.

Definition. Concurrency occurs when two RAW objects have overlapping active intervals.

3.3.1 Type 1 Concurrency

Type 1 concurrency occurs when the two RAW objects with Type 1 relationship overlap, as illustrated in figure 3.2. In Type 1 concurrency, different human resources perform different activities on the same work product at the same time. Phase overlapping (PO) is an example of Type 1 concurrency. PO overlaps consecutive development phases such as requirements and design. The requirements analysis group performs requirements analysis and passes a “partially complete” requirements specification to the design group. The design group performs architectural design based on the specification. Since the two groups perform different activities at the same time, it is an instance of Type 1 (RAW = 001) concurrency.

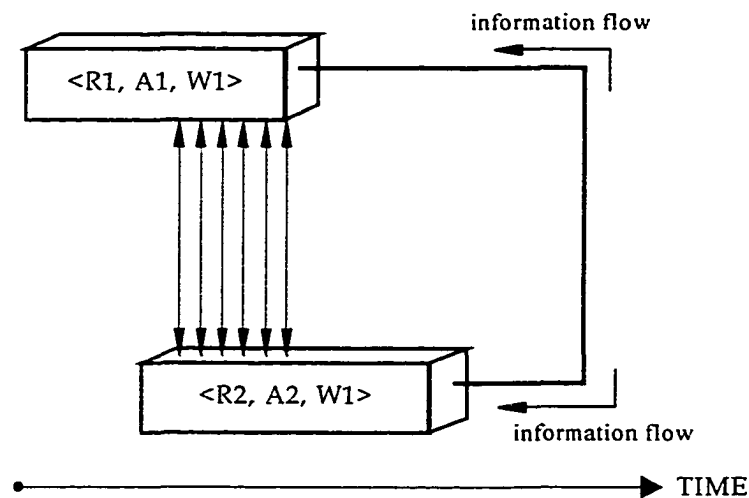


Figure 3.2. Type 1 concurrency.

3.3.2 Type 2 Concurrency

Type 2 concurrency occurs when the active intervals of two RAW objects with Type 2 relationship overlap. As depicted in figure 3.3, in Type 2 concurrency, different human resources groups (R1 and R2) perform the same activity (A2) on different work

products ($W1$ and $W2$) at the same time. In Type 2 concurrency, a system is partitioned into subsystems and assigned to different developers or teams for concurrent development. However, system decomposition can occur at different stages, such as the requirements analysis stage, the high-level design stage, and the detailed design stage. An example of requirements-stage system decomposition is Fujitsu's Concurrent Development practice [12-19]. In the development of a large-scale telecommunication software system, each release is decomposed into multiple subsystems (called enhancements) at the early stage of the development life cycle and assigned to different teams for concurrent development. We present a more detailed review of the Concurrent Development practice in section 3.4.1.

Another example of Type 2 concurrency is the traditional practice of activity concurrency in the detailed design stage, where "modules" usually are implemented by different programmers. They perform the same activity (i.e., coding) on different modules at the same time. Therefore, it is a Type 2 concurrency. However, the RAW object is defined at a lower level in which the "R" component refers to individual programmers, the "A" component refers to the coding activity, and the "W" component refers to individual program modules.

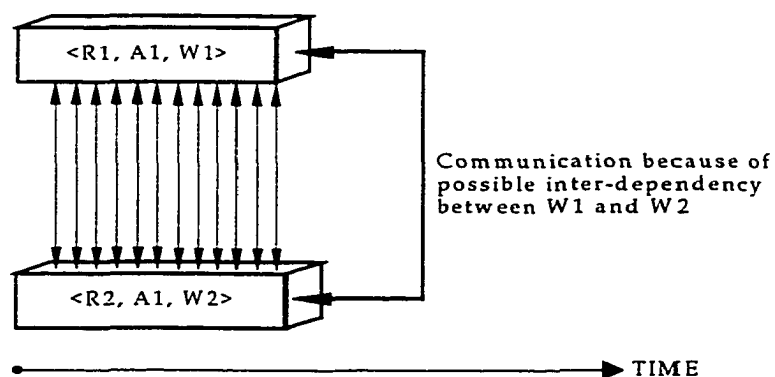


Figure 3.3. Type 2 concurrency.

3.3.3 Type 3 concurrency

Type 3 concurrency occurs when the active intervals of two RAW objects with Type 3 relationship overlap. As depicted in figure 3.4, in Type 3 concurrency, different human resources (R1 and R2) perform the same activity (A1) on the same work product (W1) at the same time. An example of Type 3 concurrency is Joint Requirements Planning (JRP) [54]. JRP involves all interested stakeholders, such as business executives, project managers, and key end-users, to define system requirements and perform high-level design. In this case, different people (with different skills and interests) perform the same activity (i.e., requirements planning and specification) on the same work product (i.e., the entire system).

Another example of the Type 3 concurrency is when two different programmers (R = 0) update (A = 1) the same program source file (W = 1) at the same time (i.e., RAW = 011). These two programmers are in a position of competing with each other. If their work is not coordinated and synchronized, their efforts might conflict with each other. This is different from the JRP where people are in a position of cooperating with one another.

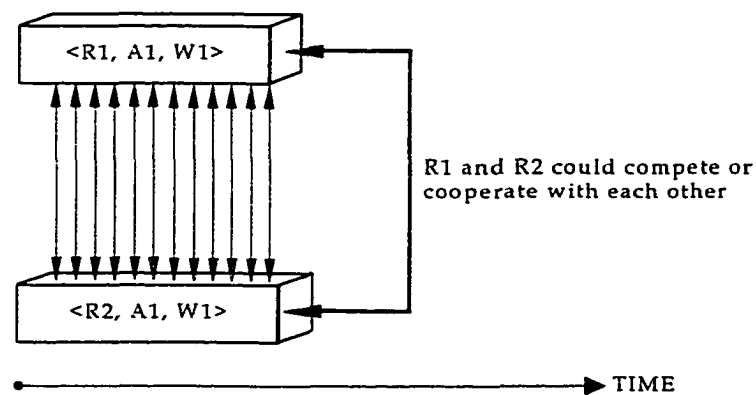


Figure 3.4. Type 3 concurrency.

3.3.4 Type 0 concurrency

Type 0 concurrency occurs when the active intervals of two RAW objects with Type 0 relationship overlap. As depicted in figure 3.5, in Type 0 concurrency, different human resources (R1 and R2) perform different activities (A1 and A2) on different work products (W1 and W2) at the same time. Type 0 (RAW = 000) concurrency is congruent with Type 2 (RAW = 010) concurrency (i.e., Synchronous Concurrent Subsystems concurrency). In the "Synchronous Concurrent Subsystems" (SCS) concurrency, different individuals or teams perform the same activity on different work products (i.e., RAW = 010). The development process is "synchronized," since they all perform the same activity (e.g., design) at the same time. However, when two individuals or teams progress at a different pace, the SCS concurrency transforms into an Asynchronous Concurrent Subsystems (ACS) concurrency. While one team is working on high-level design, the other team might progress to the detailed design or coding stage. Since they are performing different activities at the same time, therefore, the "A" component of the RAW object is changed to 0 (i.e., RAW = 000).

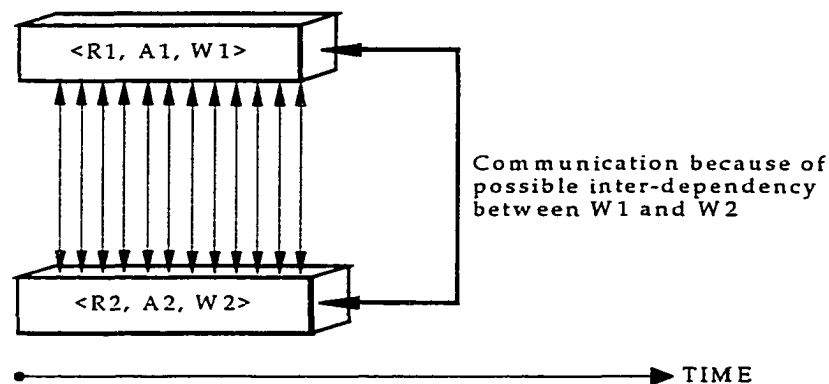


Figure 3.5. Type 0 concurrency.

We use RAW objects as a basis to classify concurrency into four types, namely, Type 0 (RAW = 000), Type 1 (RAW = 001), Type 2 (RAW = 010), and Type 3 (RAW = 011). However, the other four RAW combinations with the “Resource” component equals to 1 (i.e., 100, 101, 110, and 111) are not considered because of the following reason. The RAW model is a general model that can depict any software process models, not just the concurrent development model. When the Resource component equals to 1, it is indeed a sequential model because truly concurrency is that one single resource can perform one activity at a time.

3.4 State-of-the-Practice CSE Practices

Although concurrent engineering of software products is not a common practice in the software industry, there are a few CSE-based practices that have been used and proved effective. In this section we will review some of them and justify how the RAW model can depict them effectively.

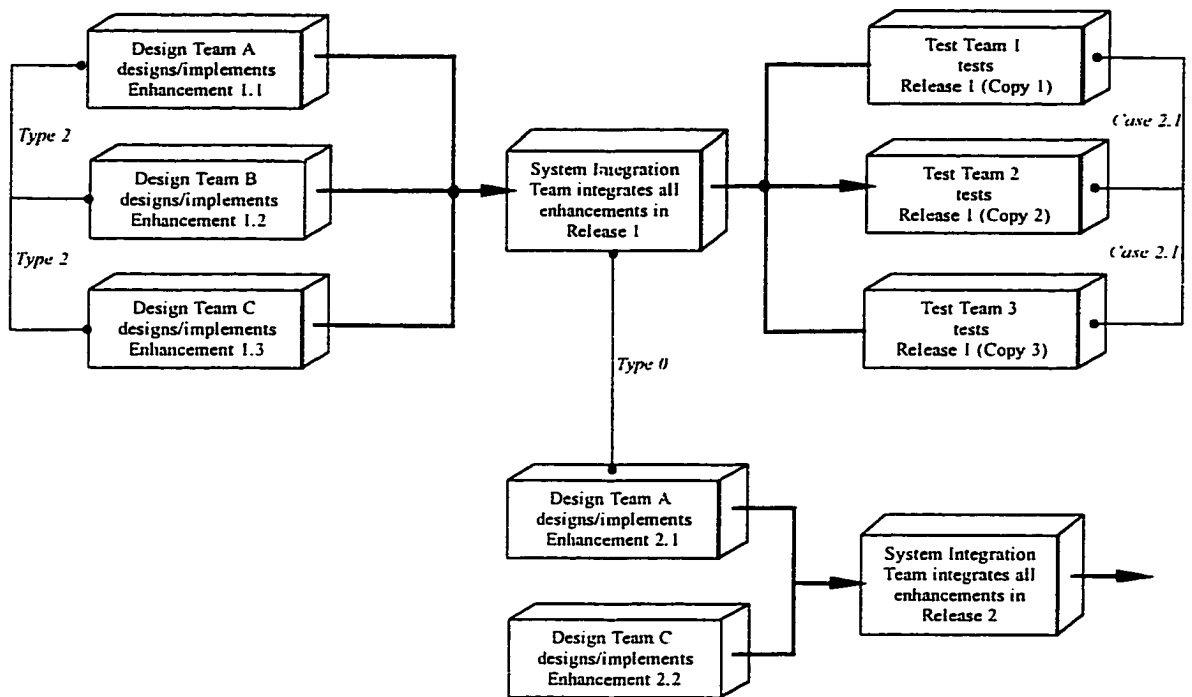


Figure 3.6. The Fujitsu concurrent development model.

3.4.1 Concurrent Development Model

One of the first successful experiences of CSE-based practice is Fujitsu's Concurrent Development Model (CDM) [12-19]. The CDM has been used in the development of a large-scale communication software system. It enables multiple small teams to work concurrently on different enhancements in a release. Multiple enhancements that are concurrently developed by different teams are incrementally integrated, tested, and delivered to the customer as a release.

A RAW-based representation of the CDM is shown in figure 3.6. As the figure shows, there are three concurrency situations in CDM, namely, Type 2, Case 2.1, and Type 0.

- **Type 2 Concurrency.** In Type 2 concurrency, different teams develop (i.e., design, implement, and perform pre-integration tests on) different enhancements at the same time. The enhancements developed by different teams usually are related to each other in certain degrees.
- **Case 2.1 Concurrency.** When all the enhancements within a release are integrated, they are distributed to different testing teams to conduct concurrent testing. Although these testings are performed on the same release, they are not on the same copy. Therefore they belong to Case 2.1.
- **Type 0 Concurrency.** When the development teams finish the development of their responsible enhancements, they move on to work on one of the enhancements of the next release. Therefore, the integration of Release 1 performed by the integration team occurs at the same time as the development of Release 2 performed by the development teams. Since they work on different work products (i.e., Release 1 and 2), their efforts belong to Type 0.

3.4.2 Concurrent Internationalization

Another CSE practice that has been used in the development of global software products is Concurrent Internationalization (CI) [65]. Traditionally, the development of global software products involves three major phases, namely, base-product engineering, internationalization, and localization. These three phases have been done sequentially in the past. After the completion of the base-product version, it must be adapted to local market conditions. Depending on the specific circumstances, this adaptation can involve minor or major changes to the base product. For example, this adaptation may require changes in user interfaces, messages, online help, language components, and even software structure. Incorporating such changes sequentially after developing the

base product requires substantial rework and therefore extends the local version's time-to-market.

Due to the ever shrinking-windows of market opportunities, software product vendors are seeking ways to reduce the time-to-market of local versions of global software products. Concurrent internationalization has been proved to be effective in this regard.

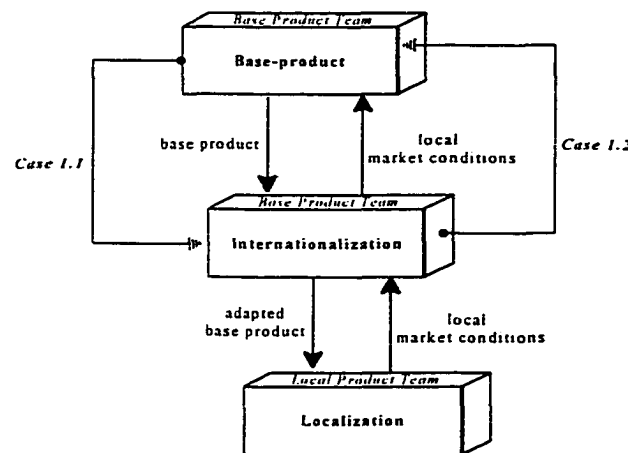


Figure 3.7. Concurrent internationalization of global software products.

Figure 3.7 shows a RAW representation of the CI practice. As illustrated in the figure, there are two concurrency situations in CI, namely, Case 1.1 and Case 1.2.

- Case 1.1 Concurrency. In this situation, two RAW objects overlap, and the information flows from upstream activities (i.e., base-product engineering) to downstream activities (i.e., local-product engineering). This is an example of phase-overlapping concurrency.
- Case 1.2 Concurrency. In this situation, two RAW objects overlap, and the information flows from downstream activities (i.e., local-product engineering) to upstream

activities (i.e., base-product engineering). By considering local market conditions and circumstances, the base-product development team is able to design a flexible software architecture that can be adapted to any local languages and market conditions.

3.4.3 Platform Development Model

The Platform Development Model (PDM) is a matrix of conceptual models for supporting platform development [44]. The objectives of the PDM are (1) to structure the development process of a family of similar products in such a way that the time-to-market of each product and the time-between-successive-products are minimized, and (2) to achieve an appropriate level of consistency across these products.

Instead of developing multiple, closely related products independently, the PDM seeks to identify and separate out common elements contained within a software product family and put them into the platform. The platform, once developed, provides a basis for value-added, differentiating features for different products within a product family.

An essential element of the PDM is the “platform and product life cycles,” as shown in figure 3.8. The major phases of the “platform life cycle” include platform requirements definition, feasibility validation, architecture definition, platform development plan, infrastructure development, code construction, and platform integration test. The “product life cycle,” which relies on the platform life cycle, has a similar underlying structure.

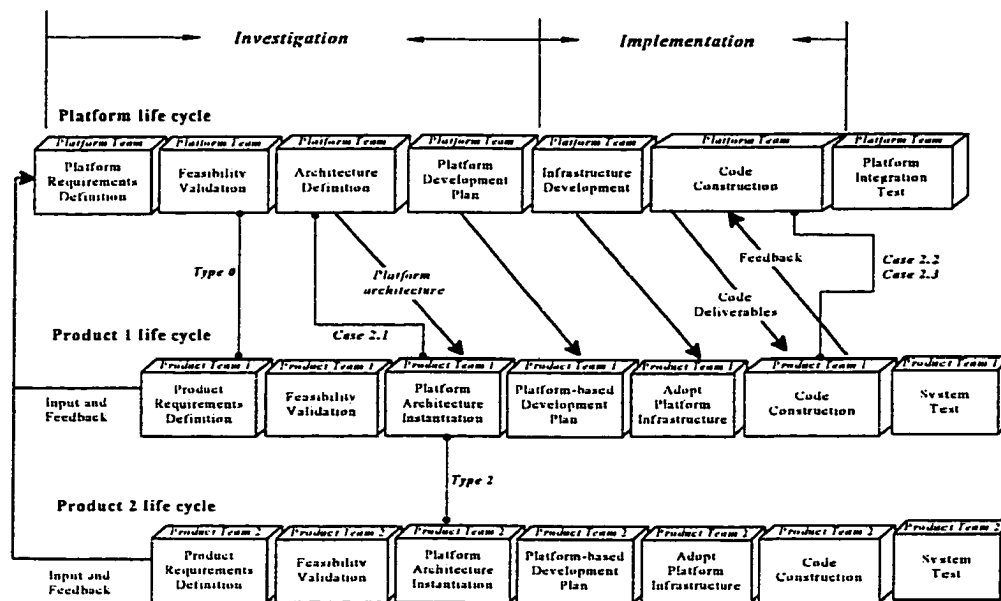


Figure 3.8. The platform development model.

There are essentially three concurrency situations in the platform development model:

- Type 0: The platform team and the product team perform different activities on different work products. For example, the platform team performs the “feasibility validation” activity on platform at the same time that the product team conducts the “product requirements definition” activity on product-unique features.
- Cases 2.2 and 2.3: The situation where the implementation phase of the platform life cycle overlaps the product implementation work is an instance either of Case 2.2 or Case 2.3 concurrency, depending on how the modules and code components in the platform and the product are related. Although Case 2.1 inter-RAW relationship exists between the platform and product teams, it is a sequential relationship. For example, the platform architecture and code components flow from the platform team to the product team after they are completed.

- Type 2: The Type 2 concurrency occurs between, for example, product team 1 and product team 2 because they perform the same activity (e.g., platform architecture instantiation) on different work products (i.e., products 1 and 2). Since the work products performed by two product teams usually do not depend on each other, Case 2.1 concurrency therefore dominates.

3.4.4 Parallel Timebox Development

Another CSE practice being used in the development of data management applications is the Parallel Timebox Development (PTD) practice [54]. As illustrated in figure 3.9, the PTD practice consists of four major phases, namely, requirements planning, user design, construction, and cutover. There are two concurrency situations in the PTD practice:

- Type 3: The first two phases (i.e., requirements planning and user design) of a PTD process involve all interested stakeholders, such as business executives, project managers, and key end-users, to define system requirements and perform high-level design. This is an example of Type 3 concurrency, that is, different function groups work on the same activity (either requirements planning and specification or high-level design) on the same work product (i.e., the entire system). After the joint requirements planning session, a central “coordinating model” is built, from which a project is partitioned. The coordination model consists of a normalized data model, a tree-structured process decomposition diagram, a process dependency diagram, data flow diagrams, and a process/data matrix.
- Type 2: In PTD, a project is decomposed into subprojects and assigned to different small SWAT (Skilled With Advanced Tools) teams for concurrent development. To manage concurrent development and make sure each SWAT team completes its share of work at approximately the same time, a rigid development time (i.e., timebox)

framework is set for all the SWAT teams. Since different teams perform the same activity (i.e., design, implementation, and unit test) on different subprojects at the same time, it is a Type 2 concurrency. The interfaces among the subsystems are defined by the coordinating model.

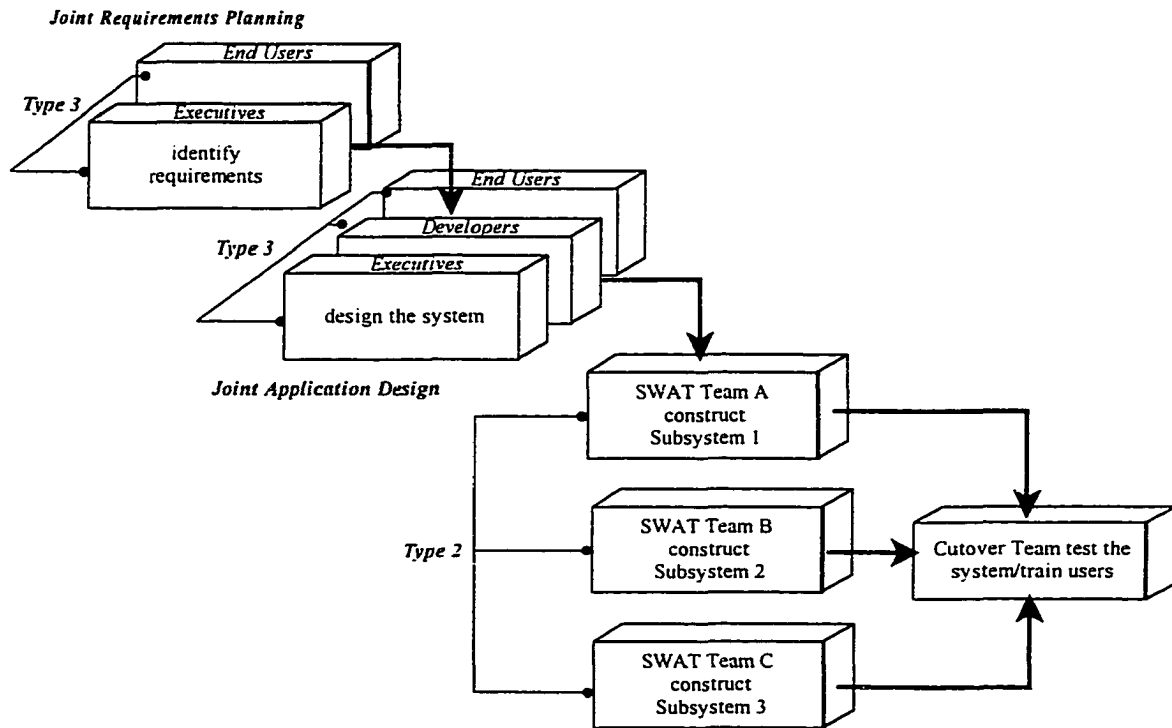


Figure 3.9. The parallel timebox development practice.

3.4.5 Hardware-Software Codesign

Another CSE approach to the development of embedded software systems (i.e., firmware) is hardware-software codesign practice. As opposed to the traditional firmware development process, in which hardware and software engineers work separately, codesign involves both communities and integrates their work. A typical design process begins with functional exploration, in which designers define a desired product's

requirements and produce a specification of the system's behavior. Hardware and software designers map this specification onto various hardware and software architectures. They then partition the functions between silicon and code and map them directly to hardware and software components. During implementation, designers either reuse or design hardware and software components. Finally, they integrate the system for prototype testing [37].

The hardware-software codesign practice is a combination of Type 3 and Case 2.3 concurrency, as illustrated in figure 3.10, depending on how hardware and software engineers work together.

- Type 3: In codesign, functional exploration, architectural mapping, and hardware-software partitioning involve both functional communities at the same time. This is an example of Type 3 concurrency in that different functional groups perform the same activity (e.g., function exploration) on the same work product (i.e., the entire system).
- Case 2.3: In the implementation stage, hardware and software engineers work on hardware and software components, respectively. This is an example of Type 2 concurrency. Specifically, since the work product performed by both communities has strong relationships, Case 2.3 applies.

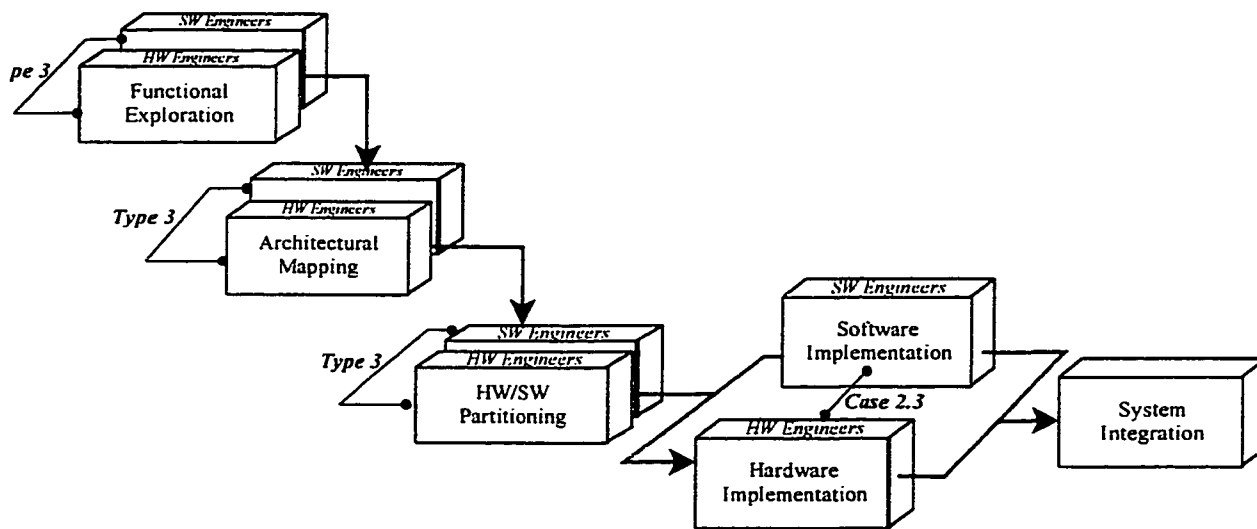


Figure 3.10. Hardware-software codesign.

3.4.6 The IPTES Approach

Incremental Prototyping Technology for Embedded real-time Systems (IPTES) is a CE approach to the development of embedded software systems [62-63]. Central to the IPTES approach is the concept of heterogeneous prototypes. A “heterogeneous prototype” is an executable system model whose different parts may be specified at different abstraction (modeling) levels, and yet they can be executed together as a total system. Models communicate through shared elements, such as data-flows, data-stores, operating system communication primitives, and procedure calls [62].

With IPTES, there could be several teams working simultaneously with different heterogeneous prototypes. A development team can use intermediate results from other teams for testing and validating their own work. Each of the development teams may use relatively abstract models of the other parts of the system as a testbed (either stubs or drivers) for their own part, yet they can proceed with developing their part at full speed by means of advancing the maturity of their part to the next abstraction level(s).

As shown in figure 3.11, the concurrent threads of development activities are organized around levels of risk. The development process includes multiple concurrent traces, where each trace corresponds to a thread of engineering activities. High-risk elements are prototyped and specified. Concurrent with the design and implementation of high-risk threads, the medium-risk elements are being specified. Later in the process, the development of activities of different risk-level proceeds concurrently. They are incrementally integrated, installed, and put into use.

Concurrent engineering can take place at the level of concurrent threads, or it may take place at a subsystem level (i.e., work for each subsystem may contain concurrent threads) [62]. IPTES is an example of Type 0 concurrency, since different teams perform different activities (i.e., heterogeneous prototyping) on elements of different risk-level.

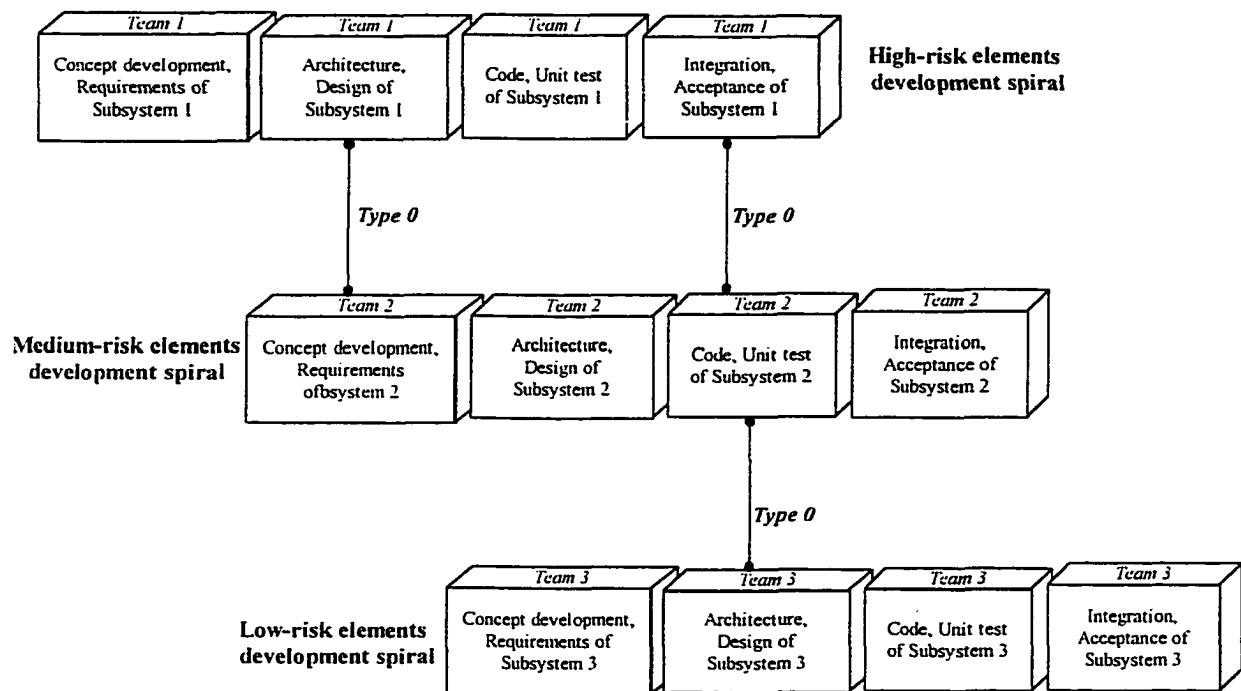


Figure 3.11. The IPTES approach.

3.4.7 Microsoft Daily Build Process

Another CSE approach to the development of commercial software products is Microsoft's Daily Build (DB) process. The DB process begins with a "vision statement" outlining the goals of a new product. An initial set of product features is identified and priority-ordered based on their importance in supporting end-users' activities. The list of prioritized features is then partitioned into three to five feature sets that small teams can develop in a few months.

The DB process enables multiple *feature teams* to work in parallel. Each feature team is responsible for a specific set of product features end-to-end from feature specification, design and coding, to feature integration and testing. With the DB process, specifications, development, and testing are carried out in parallel. However, the teams synchronize their work by building the product and finding and fixing errors on a daily and weekly basis. This is achieved by maintaining a shared master version of the implemented product. Developers have the freedom to evolve design and implementation of their responsible features; however, they must check in their work at least twice a week. As illustrated in figure 3.12, there are essentially two concurrency situations in the DB practice, namely Type 1 and Type 2.

- Type 1: Each feature team usually consists of similar number of developers and testers. The development and testing are done in parallel. Here, developers and testers perform different activities on the same feature. Developers are responsible for feature specification, design, and implementation. The testers prepare test plans and design test cases based on the preliminary information about the specification and design provided by the developers. The detected defects are fed back to the developers for revision and improvement. The ongoing concurrent activities between developers and testers is an example of Type 1 concurrency (different people working on different activities on the same work product).

- **Type 2:** This inter-RAW relationship occurs between developers of a feature team, developers of different teams, and different feature teams. For example, the situation where Developers 1 and 2 perform the same activity (i.e., design/implement) on different work products (Features 1 and 2) at the same time is an instance of Type 2 concurrency. All three cases are possible, depending on how Features 1 and 2 are related to each other.

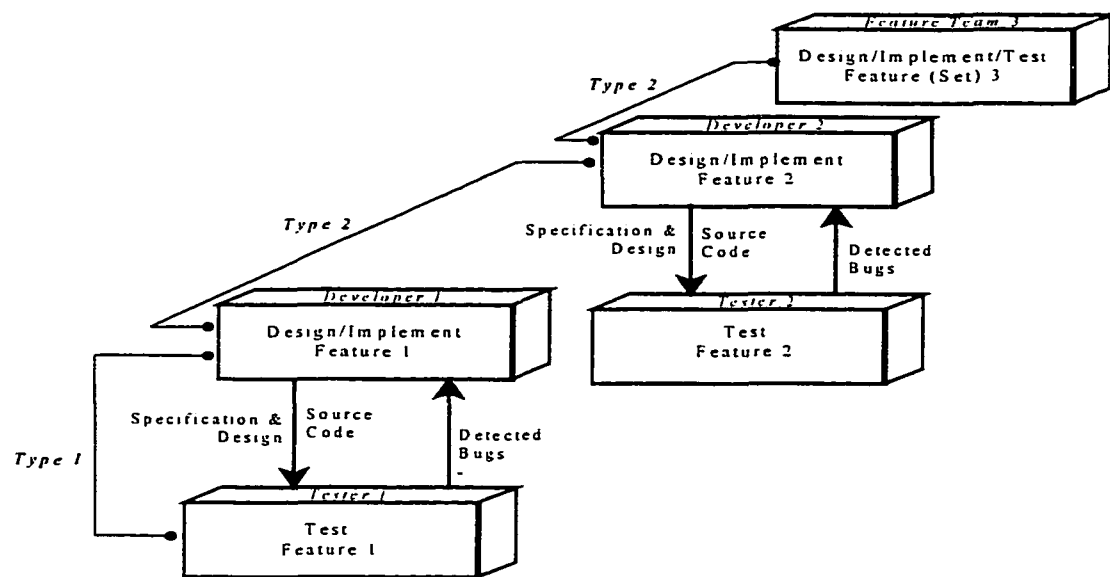


Figure 3.12. The Microsoft daily build process.

CHAPTER 4

A SYSTEM DYNAMICS MODEL

4.1 Introduction

This chapter presents the proposed concurrent software engineering system dynamics simulation model CSE-SD. Our purpose is to gain insight and understanding about the impact of CSE on software project development with a focus on project cost and development cycle time. CSE-SD is drawn from extensive literature review and interviews with software project managers.

We will then use the simulation model as a research vehicle to investigate a set of preliminary questions. CSE-SD can answer numerous software project management questions, such as “Will an increased degree of concurrency shorten project development cycle time?” The results of other important questions are presented in chapters 6 and 7.

In the next section we will examine the benefits and problems of each type of concurrent software engineering and their dynamic implications. They are represented as a set of cause-effect feedback relationships. These feedback relationships serve as the foundation of CSE-SD. In section 4.3, we present an overview of the overall model structure and explain the main functions of each model component. A detailed specification of the model, including formal model equations, is included in appendices A and B.

4.2 Dynamics of Concurrent Software Engineering

In this section we describe the underlying cause-effect feedback structures of the CSE-SD model. The feedback structures aim to address the issues of the four types of concurrency discussed in section 3.3.

4.2.1 Phase Overlapping

Phase overlapping in hardware manufacturing industry has shown a strong correlation between the degree of phase overlapping and shorter development life cycle. This approach, however, is not well adopted in the software industry, since software development has a “soft” front end. Requirements changes of 25% or more are not unusual [22]. Beginning the high-level design activities before the requirements definition has stabilized increases the risk that changing specifications will require redesign, and the cost of reworking a stage can be exorbitant [21].

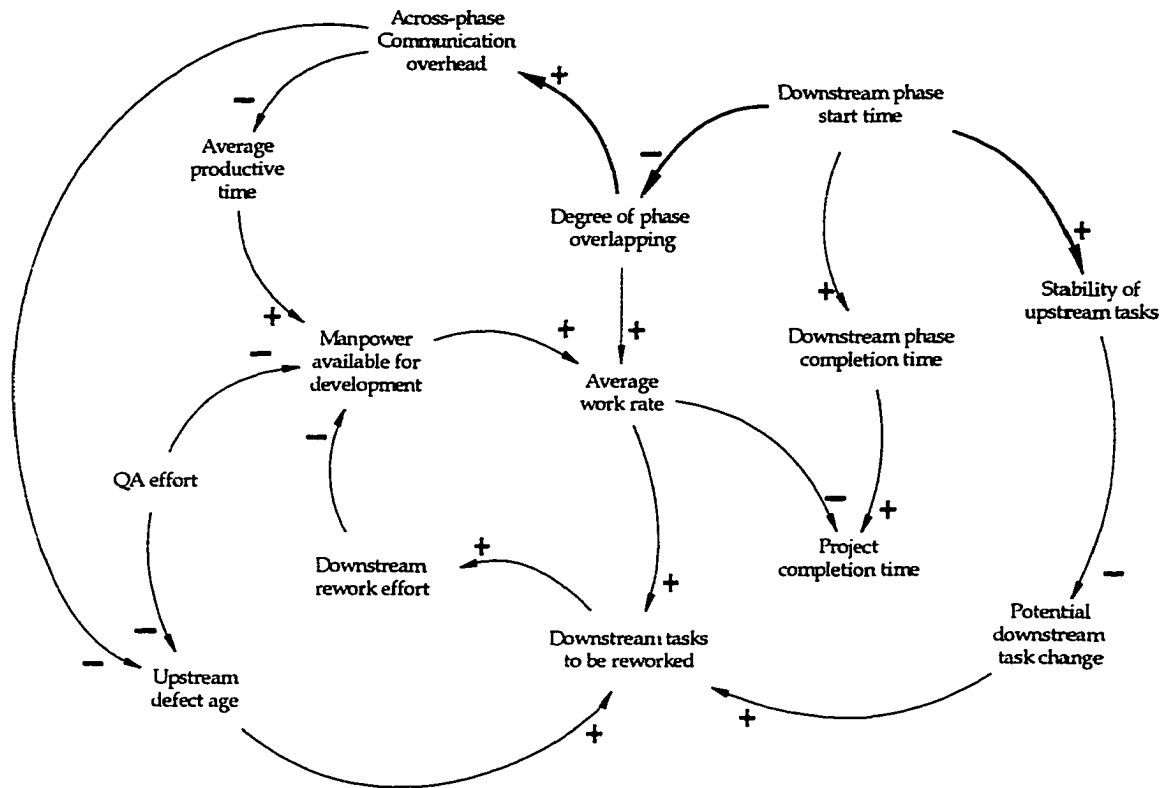


Figure 4.1. Dynamics of phase overlapping.

Figure 4.1 shows the cause-effect dynamics of attempting the *phase overlapping* software development approach. Phase overlapping happens when project development starts a downstream phase before the upstream phase is completed. A positive effect is that, by starting earlier, the downstream phase can complete earlier. Therefore, the project can be completed earlier. Another positive effect is that more work can be done at the same time, therefore, the overall average work rate is increased. As a result, the project can be completed earlier. The two positive effects of starting downstream phase early are depicted in the following two causal links (CLs):

Downstream phase start time +> Downstream phase completion time +>

Project completion time (CL 6)

Downstream phase start time -> Degree of phase overlapping +> Average

work rate -> Project completion time (CL 7)

Note that A +> B (A -> B) represents A and B change in the same (opposite) direction. For example, increasing A will incur an increase of B.

Instead of waiting for the completion of the upstream phase, downstream engineers need to use preliminary information from the upstream phase. This has a negative effect on project completion time, as shown in CL 8. Since the exchanged information is not yet stable, any changes to the exchanged information must be incorporated in the downstream phase. The more unstable the information being used by downstream engineers, the more potential changes to downstream tasks can be expected. The unexpected increase of the downstream rework tasks will consume part of the person-day resource originally allocated to planned development tasks, which leads to the decrease in the overall average work rate. As a result, the project completion time is prolonged.

Downstream phase start time +> Stability of upstream tasks -> Potential
downstream task change +> Downstream tasks to be reworked +>

Downstream rework effort -> Manpower available for development +>

Average work rate -> Project completion time (CL 8)

The negative effect of using unstable information is exacerbated when downstream tasks are performed at a faster pace (i.e., higher average work rate). This leads to more downstream rework tasks to be generated. More rework tasks requires more rework effort. Therefore, the manpower resource originally allocated to planned development tasks is reduced. The end result is that project completion

time being delayed even further. The negative effect of starting downstream phase early is depicted in the following causal link:

Downstream phase start time -> Degree of phase overlapping+> Average work rate +> Downstream tasks to be reworked +> Downstream rework effort -> Manpower available for development +> Average work rate -> Project completion time (CL 9)

Phase overlapping increases the need for engineers in different phases to communicate with each other. Two-way, high band-width information flows are needed to keep the process from getting “out-of-sync” and to compress the time between occurrence and detection of problems [21]. The negative effect of starting the downstream phase early is depicted in the following causal link:

Downstream phase start time -> Degree of phase overlapping+> Across-phase communication overhead -> Average productive time +> Manpower available for development +> Average work rate -> Project completion time (CL 10)

Using defective information from the upstream phase regenerates more downstream defects. The longer the defective information remains undetected, the more the downstream defects will be amplified. Therefore, the time between occurrence and detection of the defects in the exchanged information (Upstream Defect Age) has an impact on the amount of downstream tasks that need to be reworked. Communication across two phases, although helpful to detect problems early, takes away from the staff’s productive time (Average Productive Time). A decreased average productive time means that decreased manpower will be available for planned development tasks. As a result, the average work rate is decreased, which leads to the project completion time being extended. QA activities have similar effects. They help to detect defects early, before they are regenerated and amplified. The effects of

effective across-phase communication and QA on project completion time are depicted in the following three causal links:

Downstream phase start time -> Degree of phase overlapping+> Across-phase communication overhead -> Upstream defect age +> Downstream tasks to be reworked +> Downstream rework effort -> Manpower available for development +> Average work rate -> Project completion time (CL 11)

QA effort -> Upstream defect age +> Downstream tasks to be reworked +> Downstream rework effort -> Manpower available for development +> Average work rate -> Project completion time (CL 12)

QA effort -> Manpower available for development +> Average work rate -> Project completion time (CL 13)

4.2.2 Synchronous Concurrent Subsystems

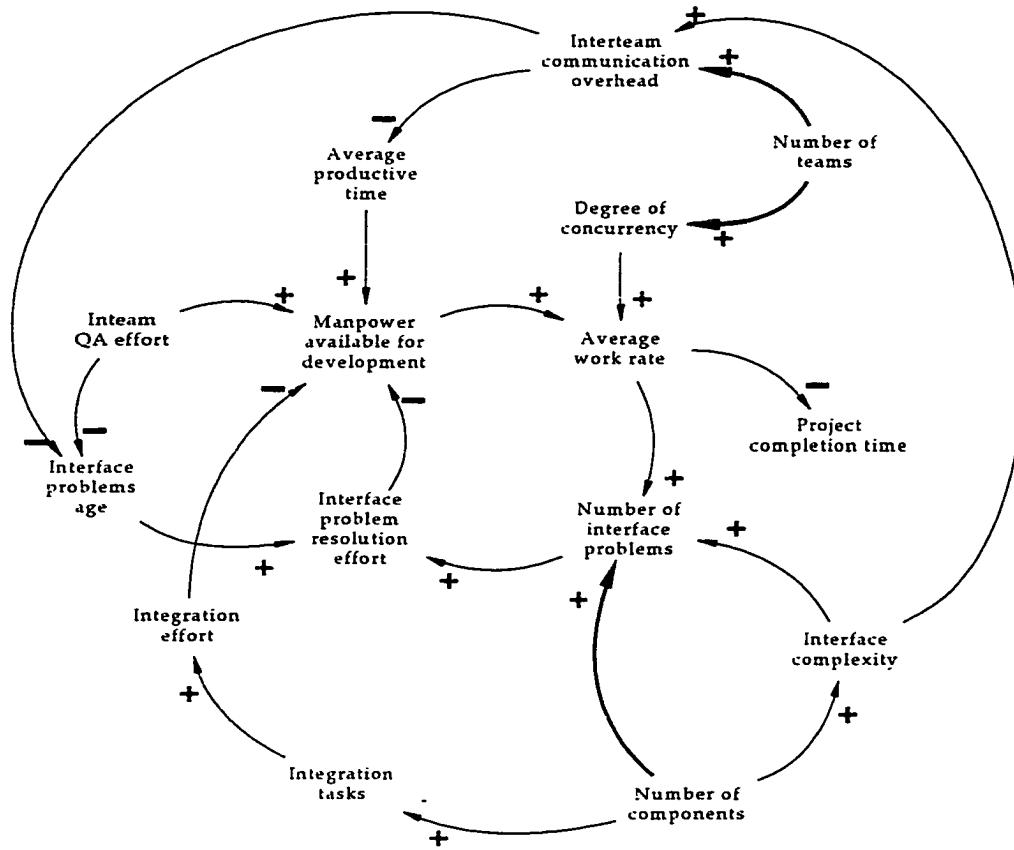


Figure 4.2. Dynamics of synchronous concurrent subsystems.

Synchronous concurrent subsystems (SCS) is a common practice in the software development industry. However, it normally is practiced in the detailed design stage, where modules with well-defined interfaces (ideal situation) are assigned to different programmers for concurrent implementation. Recently, software development companies have been seeking ways to practice concurrent subsystems development in the early stages of the development life cycle, such as requirements analysis and high-level design ([14], [34], [44]). The benefits and potential risks of the

synchronous concurrent subsystems development approach are examined in this section.

Figure 4.2 shows the dynamics of the synchronous concurrent subsystems development approach. Two key milestones in a SCS-based project are problem decomposition and synchronization/integration.

Large-scale software systems must be decomposed into components, so they can be assigned to multiple teams and/or individuals for concurrent development. The total number of components, their contents, and sizes are important issues. If a system is decomposed into more components, they then can be assigned to more development teams. More concurrent development teams means more tasks are being done at the same time (increased degree of concurrency). The overall average work rate is increased, and as a result, the project completion time is reduced. The effects of increasing the number of concurrent teams are depicted as the following causal link:

Number of teams +> Degree of concurrency +> Average work rate -> Project completion time (CL 14)

There is, however, a negative effect, as well, when the number of concurrent teams is increased, as depicted in CL 15. As the number of teams increases, more inter-team communication traffic is expected, especially when the system is not well partitioned (i.e., high-interface complexity). Therefore, staff members' average productive time is decreased, which leads to the decrease of available manpower resource for planned development tasks. The end result is that project completion time is delayed even further.

Number of teams +> Interteam communication overhead -> Average productive time +> Manpower available for development +> Average work rate -> Project completion time (CL 15)

As the number of components increases, the interfaces among components become more complicated. Higher interface complexity has two negative effects on project schedule, as depicted in CL 16. First, a complex interface incurs more communication overhead among development teams. As project staff members spend more time communicating with other teams, the time they can spend on development work is decreased. Decreased productive time means decreased manpower is available for planned development tasks. As a result, the overall average work rate is decreased, and the project completion time is extended.

Interface complexity +> Interteam communication overhead -> Average productive time +> Manpower available for development +> Average work rate -> Project completion time (CL 16)

The second effect of a complex interface is that interface problems are more likely to happen, and as the number of components increases, the effect becomes more serious. Interface problems have to be resolved sooner or later. More interface problems mean more interface problem resolution effort is needed. As manpower is allocated to resolve interface problems, the available manpower available for planned development tasks is decreased. The overall average work rate also decreases. As a result, the time to project completion is extended. The effects of a complex interface are depicted in the following two causal links:

Number of components +> Interface complexity +> Number of interface problems +> Interface problem resolution effort -> Manpower available for development +> Average work rate -> Project completion time (CL 17)

Number of components +> Number of interface problems +> Interface problem resolution effort -> Manpower available for development +> Average work rate -> Project completion time (CL 18)

Another negative effect of increasing the number of components is the

increased number of integration tasks. More components mean more tasks have to be integrated. System integration takes away part of the manpower allocated to planned development tasks. As a result, the overall average work rate is decreased, and the project completion time is extended, as illustrated in CL 19.

Number of components +> Integration tasks +> Integration effort ->

Manpower available for development +> Average work rate -> Project
completion time (CL 19)

Concurrent development without synchronization and coordination among concurrent development teams throughout the project life cycle can result in interface problems that surface at the end, when the components are integrated. For example, in firmware development, delaying the integration of hardware and software until the first testable hardware prototype is troublesome for several reasons. Engineers have little time to correct design problems, and fixes are more costly than they are earlier in the design process. Options for revisions are much more limited; because of the rigidity of the hardware, design changes usually are made in the software, at the expense of system performance [21].

Effective communication between engineers of two different teams and the quality of the exchanged information both help to shorten the time between the introduction and the detection of an interface problem (i.e, interface problem age). If an interface problem is not detected close to the time it is introduced, then more interface problems will be regenerated as it flows into downstream phases. The later an interface problem is detected, the more interface problems will be amplified, which, in turn demands more interface problem-resolution effort. As a result, the planned development tasks are delayed, and the overall average work rate is reduced, which leads to an extended project completion time, as depicted in CL 20.

Interteam communication -> Interface problem age +> Number of interface problems +> Interface problem resolution effort -> Manpower available for development +> Average work rate -> Project completion time (CL 20)

Besides frequent communication among concurrent teams, periodic interteam QA activities (e.g., specification and design reviews) help to locate interface problems early, before they are amplified when they flow into subsequent phases. Interteam QA, although helpful to reducing the interface problem age, nonetheless takes away staff members' productive time. The reduced average productive time means less manpower will be available for planned development tasks. Therefore, the overall average work rate is reduced. As a result, the project completion time is extended.

Interteam communication -> Interface problem age +> Number of interface problems +> Interface problem resolution effort -> Manpower available for development +> Average work rate -> Project completion time (CL 21)

4.2.3 Asynchronous Concurrent Subsystems

The Asynchronous Concurrent Subsystems (ACS) concurrency is congruent with the Synchronous Concurrent Subsystems (SCS) concurrency. In SCS, different teams perform the same activity on different work products. The development process is "synchronized," since different subteams perform the same activity (e.g., design) at the same time. However, when two subteams progress at a different pace, the SCS concurrency transforms into the ACS concurrency. ACS is an example of "Type 0 (000)" concurrency because different teams ($R = 0$) perform "different" activities ($A = 0$) on different work products ($W = 0$) at the same time.

Although the development is not synchronous (i.e., each subteam evolves its design at different speed), the subteams' work must be integrated at the end of the

project. Therefore, it is important to know how to control the development progress of each team, to be sure they will complete their share of work on time.

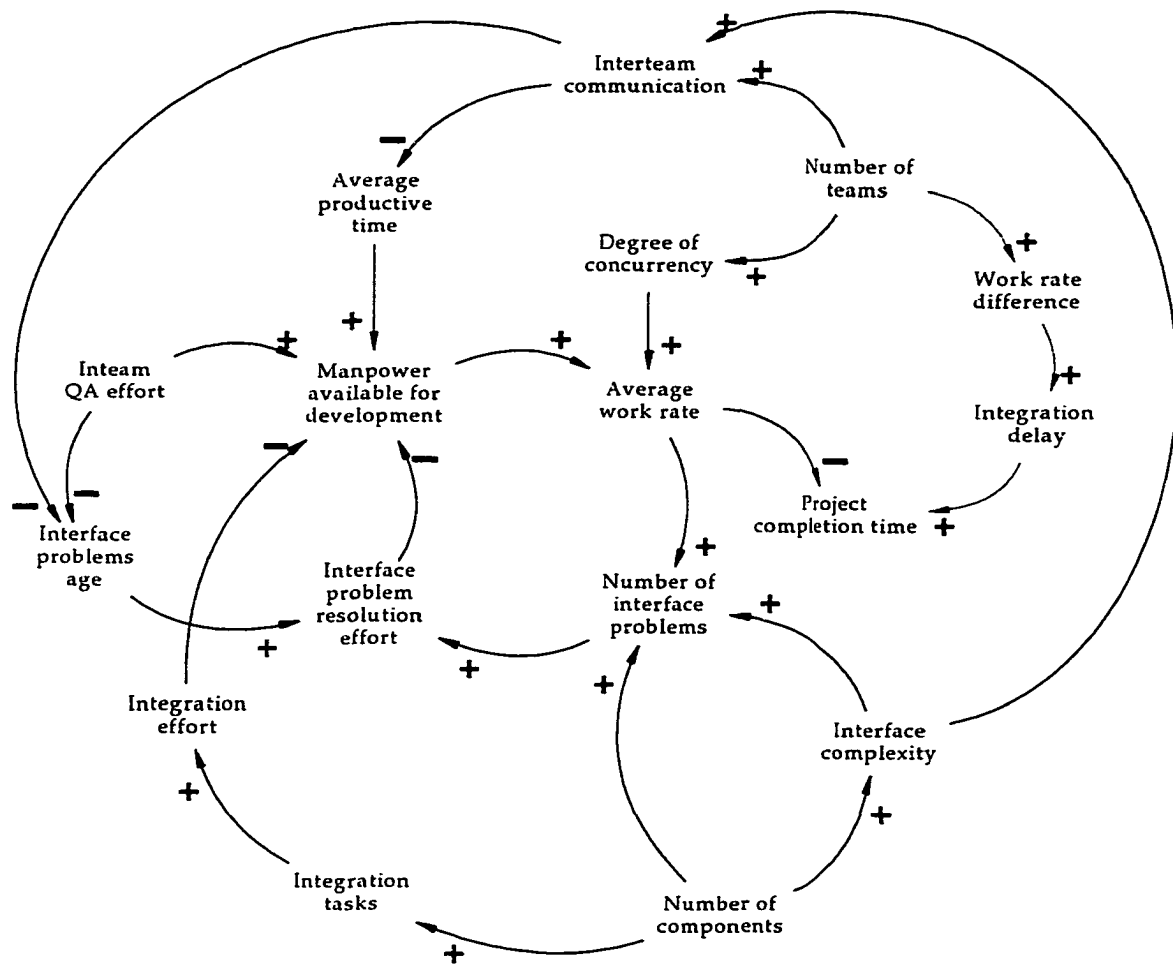


Figure 4.3. Dynamics of asynchronous concurrent subsystems.

Increasing the number of development teams will increase the degree of concurrency, but it also increases the probability that the work rate of each team will be deviated from each other. Faster teams have to wait for slower teams to complete

before their work can be integrated. The delay of integration prolongs the project completion time, as the following causal link shows:

Number of teams +> Work rate difference +> Integration delay +> Project completion time (CL 22)

4.2.4 Cross Function Integration

Concurrent engineering (CE), as practiced in the hardware manufacturing industry, is an instance of Cross Function Integration (CFI) concurrency. CE integrates expertise in multiple functions by forming a cross-functional team that involve engineers from different functional areas: hardware and software engineering, marketing, process engineering, business development, customer engineering, and manufacturing. Each member is involved in every stage of the product cycle [67]. In a cross-functional team, engineers from different functional disciplines perform the same activity on the same work product at the same time. Therefore, CFI is an example of “Type 3 (011)” concurrency. The cause-effect relationships of the Cross Function Integration (CFI) are shown in figure 4.4.

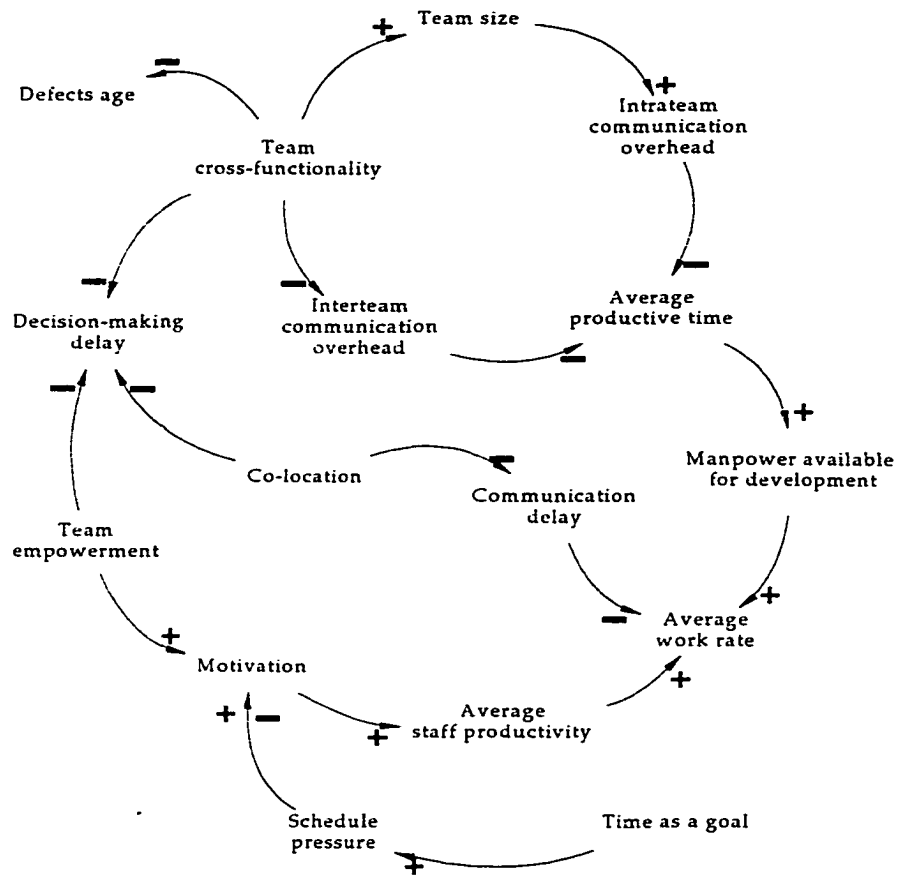


Figure 4.4. Dynamics of cross function integration.

In CE, the key ingredient is teamwork [67]. Product development time is reduced through many teamwork-related mechanisms, including (1) cross-functional teaming; (2) empowerment of decision-making authority; (3) co-location of core team members; and (4) setting time as a goal [85].

Cross-functional development teams, formed at the start of a project, facilitate the communication of product requirements and constraints among the functional groups. This enables early problem identification, better cross-functional

coordination and faster decision-making [85]. Teams composed of members from different technical areas have been shown to be better at exploring design decisions in breadth by posing alternatives and constraints and by challenging assumptions [33]. Being able to quickly make high-quality decisions is a critical factor in fast product development. Decision-making is made easier with a cross-functional team, because the primary information providers and decision-makers are part of the team [85]. The effects of forming multi-disciplinary teams are depicted in the following four causal links:

Team cross-functionality -> Interteam communication overhead +> Average productive time +> Manpower available for development +> Average work rate -> Project completion time (CL 23)

Team cross-functionality -> Decision-making delay -> Average work rate -> Project completion time (CL 24)

Team cross-functionality -> Defect age +> Number of tasks to be reworked +> Rework effort -> Manpower available for development +> Average work rate -> Project completion time (CL 25)

Team cross-functionality +> Team size +> Intrateam communication overhead -> Average productive time +> Manpower available for development +> Average work rate -> Project completion time (CL 26)

Personnel factors have the greatest potential to shorten software project schedule across a variety of projects [57]. Motivation is undoubtedly the single greatest influence on how well people perform. Most productivity studies have found that motivation has a stronger influence on productivity than any other contributing factor [22]. The sense of empowerment has a motivating effect on staff members.

Empowerment, or downward delegation of decision-making power, has motivational impacts. Instead of waiting for senior management's approval, project team

members are empowered to make and implement their own decisions. Empowerment results in an increased motivation to do things better and faster. The more autonomy people have, the greater the sense of personal responsibility they tend to feel for the outcome of their work [57]. They own the schedule, but they feel the pressure of the market. This pressure causes them to reach for tools on their own [24]. Empowerment motivates engineers to work harder, especially under schedule pressure. The effects of empowerment are depicted as the following three causal links:

Team empowerment -> Decision-making delay -> Average work rate -> Project completion time (CL 27)

Team empowerment +> Motivation +> Average staff productivity +> Average work rate -> Project completion time (CL 28)

Team empowerment +> Schedule pressure +> Average work rate -> Project completion time (CL 29)

Locating project team members close together can speed up development by facilitating communication and decision-making [85]. It is costly to collect and disseminate information among distributed development teams. Dividing the process into multiple teams may block the smooth flow of information and development knowledge [19]. The effects of co-location are depicted as the following three causal links:

Co-location -> Communication delay -> Average work rate -> Project completion time (CL 30)

Co-location -> Decision-making delay -> Average work rate -> Project completion time (CL 31)

Goal setting is another key to achievement motivation. Setting time as a goal speeds up the development process [85]. The experience of consumer products

illustrates that competitive benchmarking and focusing on reducing the time needed to realize new products can drive significant process improvement.

Setting too aggressive a goal, however, has negative effects on project performance. As schedule pressures increase, commitment will increase to some point, and then decline as motivation declines due to overwork or disillusionment with the project or the organization [59]. You should keep commitment up by maintaining a slight-to-modest schedule pressure using deadlines and setting goals that challenge work groups without exhausting them.

Time as a goal +> Schedule pressure +> (or ->) Motivation +> Average staff productivity +> Average work rate -> Project completion time (CL 32)

4.3 Model Structure

As shown in table 4.1, the proposed concurrent software engineering system dynamics (CSE-SD) model consists of five subsystems, namely, *Work Flow*, *Defects and Rework*, *Human Resource*, *Manpower Allocation*, and *Manpower Needed*, and four other independent sectors, *Planning*, *Project Control*, *Interteam Interactions*, and *Project Scope Change*. CSE-SD is implemented in *ithink analyst* [43] software package. An overview of the model is shown in figure 4.5. The main functions of each model component and their relationships are described below.

Table 4.1. Major components of the CSE-SD model

SUBSYSTEM	SECTOR
HUMAN RESOURCE	Work Force
	Staff Productive Time
	Staff Productivity
WORK FLOW	Requirements Work Flow
	Development Work Flow
	System Integration and Test (SIT)
DEFECTS AND REWORK	Requirements Defects and Rework
	Development Defects and Rework
MANPOWER ALLOCATION	Requirements Manpower Allocation
	Development Manpower Allocation
	SIT Manpower Allocation
MANPOWER NEEDED	Requirements Manpower Needed
	Development Manpower Needed
	SIT Manpower Needed
INDEPENDENT SECTORS	Planning
	Project Control
	Interteam Interactions
	Project Scope Change

The *Human Resource* subsystem consists of three sectors: *Work Force*, *Staff Productive Time*, and *Staff Productivity*. The *Work Force* sector keeps track of the number of software engineers currently working on the project. We divide work force into

two categories, namely, new staff and experienced staff, for three reasons. First, new staff members usually are less productive due to their lack of project experience and knowledge. Second, new staff members usually spend most of their time in training and orientation right after they are brought into the project. Training also consumes part of the experienced staff members' productive time. The third reason is that new staff members usually are prone to commit more errors than experienced staff members.

The *Staff Productive Time* sector monitors the staff time resource. It breaks down the project staff's daily time into two main categories: project time and slack time. Project time is the time that staff members spend on project-related activities, including development, training, and project-related communication. It is further divided into three different categories: productive time, training time, and communication time. Productive time includes the time that staff members spend directly on development activities, such as requirements specification, design, coding, testing, QA, and rework. Training time keeps track of the time that project staff spends in training per day, including the time spent both by experienced staff members and new staff members in training-related activities.

Communication time captures the amount of time that project staff spends in communicating with other members within a team and across teams. A well-partitioned project usually has higher communication traffic within a team than across teams. Slack time captures the time that project staff spends in non-project-related events, such as coffee breaks, sickness, and so forth. Project staff overtime also is monitored.

The *Staff Productivity* sector determines the average production rate of project staff members (i.e., number of tasks performed per staff per unit time). Although numerous factors could affect staff members' production rate, we focus on four

factors that have dynamic implications, namely, work force mix, learning effect, schedule pressure, and staff exhaustion level. The average staff production rate at any point in time is determined by multiplying the “nominal staff production rate” (defined as the production rate of the experienced staff working under normal condition, that is, there is no schedule pressure and they are not exhausted) by the four factors. Schedule pressure, learning effect, and work force mix (more experienced staff members equate to a larger production rate) have a positive impact on staff production rate, while staff exhaustion level has negative effects.

The *Work Flow* subsystem models the software production activities, ranging from requirements specification, software design, coding, to system integration and test. It consists of three sectors, and each sector models the software production process of the three phases modeled in CSE-SD, namely, requirements, development (including design and coding), and system integration and test.

The *Defects and Rework* subsystem models the generation, detection, and rework of detected defects. Three categories of defects are of concern: requirements specification defects, development defects, and bad fixes. One important reason to classify defects into these three categories is that different types of defects require different costs to fix. Defects originated in upstream phases, such as requirements, will flow into downstream phases if not detected. For example, a design based on inconsistent requirements specification is defective, no matter how perfect the design is.

The *Manpower Allocation* subsystem allocates the planned project effort to different software engineering activities, including requirements specification, development, QA, defect correction, and system integration and test.

The *Manpower Needed* subsystem determines, at any stage of the development life cycle, the effort perceived still needed to complete the project, including effort

needed for requirements specification, specification QA, specification defect correction (determined in the *Requirements Manpower Needed* sector), development, development QA, development defect correction (determined in the *Development Manpower Needed* sector), system integration, system test, and defects found in the system test phase (determined in the *SIT Manpower Needed* sector).

The amount of effort perceived still needed to complete the project is determined based on how well project staff performed in the past. In the early stage of the development life cycle, project staff members usually do not know exactly how productive they are. The perception of their productivity simply is their planned productivity. However, when the project progresses to the end, they begin to realize how productive they are. Therefore, their perception of their productivity approaches their actual productivity. The total effort perceived still needed to complete the project is fed into the *Project Control* sector to decide whether or not to adjust project effort, schedule, work force, or all three, if needed.

The *Planning* sector is the entry point to the CSE-SD model. Its main functions are to compute and distribute the estimated project effort to different phases of the software development life cycle. Prior to initiating a software development project, managers must estimate three things before the project begins: how long it will take, how much effort will be required, and how many people will be involved [61]. Accurate estimation of the project effort, schedule, and required work force, however, relies on an accurate estimate of the product size. To run the model, the simulator must provide a value for the initial estimate of the project size. CSE-SD calculates project effort, schedule, and expected work force based on the COCOMO cost estimation model [22-23].

The *Project Scope Change* sector models the change in the scope of a software project. Reasons that cause the project scope to change include incomplete and

conflicting requirements specifications, requirements uncovered due to project underestimation, and new requirements. Unplanned requirements, when discovered and incorporated into the project plan, will cause part of the existing development tasks to be deleted or modified, and new tasks will be added.

The *Interteam Interactions* sector deals with the interteam issues that result from multiple concurrent activities. It models the generation, detection, and resolution of problems and issues caused by multiple concurrent teams that could be avoided if done by a single team. For example, multiple teams working on related subsystems may disrupt the system integrity. In requirements specifications, for example, this can cause inconsistent or incomplete specifications. In design and implementation, simultaneous updates to a single module may violate that module's consistency [14].

Undetected interteam problems tend to propagate through succeeding tasks that build on one another, such as through design and coding tasks built on inconsistent requirement specifications. Resolution of detected interteam problems leads to the rework of some of tasks.

The *Project Control* sector monitors and controls a software development project. It combines the effort perceived still needed to complete the project from the three *Manpower Needed* sectors and compares it with the planned development effort that is remaining. Corrective actions are taken when these two measures deviate significantly from each other. Corrective actions that usually are taken by software project managers are modeled in CSE-SD, including modifying the planned project effort and schedule, changing the planned work force, adjusting the planned QA and testing effort, or a combination of the three.

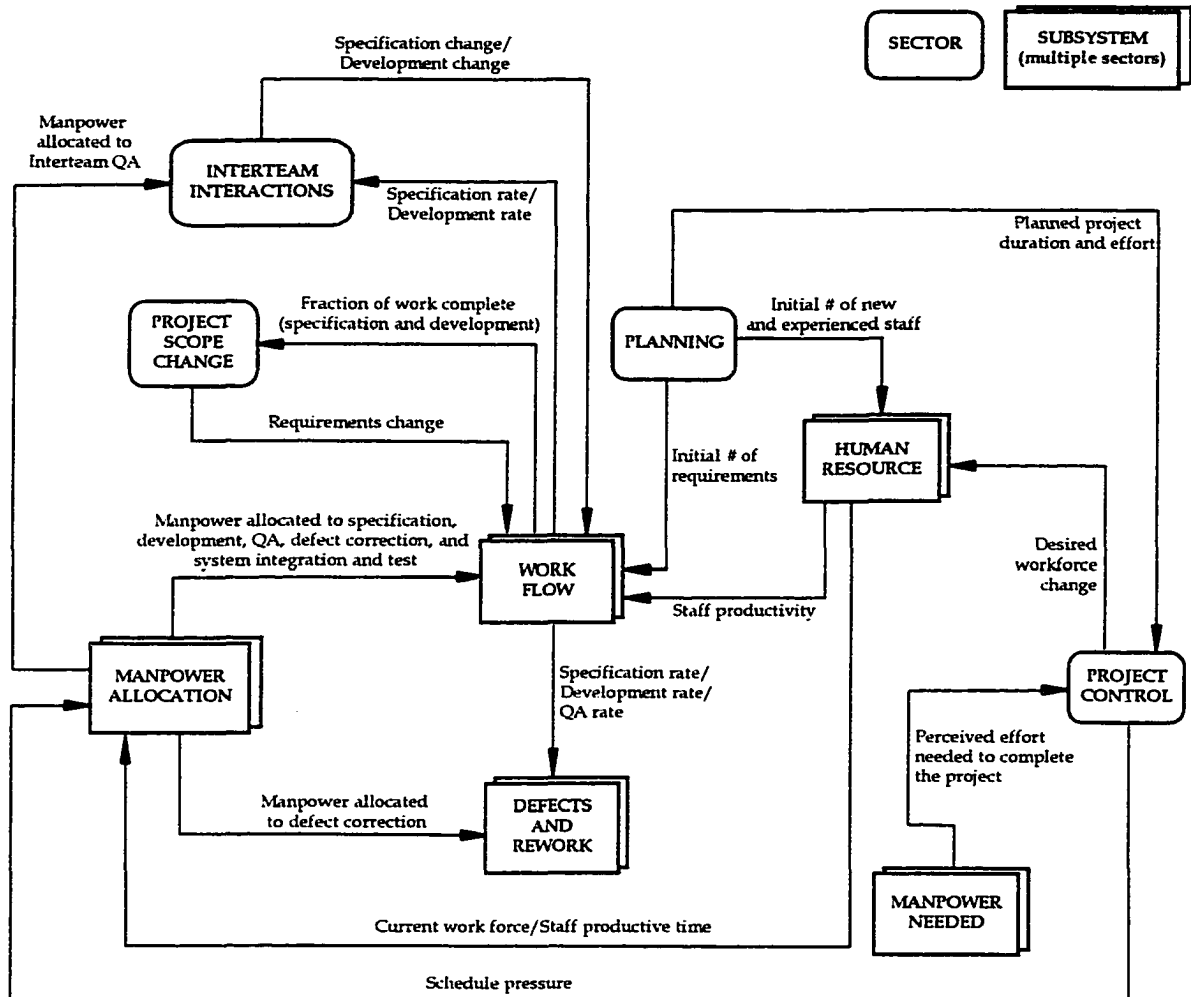


Figure 4.5. Overview of the CSE-SD model.

4.4 Comparison with Other Related SD Models

In this section, we review four related software project system dynamics models, including Abdel-Hamid and Madnick [2-10], JPL [48-50], Madachy [52-53] and Collofello and Tvedt [79-80], and compare CSE-SD with each one of them.

4.4.1 Abdel-Hamid and Madnick

The Abdel-Hamid and Madnick (AHM) software project system dynamics model represents one of the first efforts in this area. The AHM model covers important issues of software project management, presents numerous system dynamics modeling strategies, and includes quantitative data that motivate us to employ the system dynamics approach to study the impact of concurrent software engineering. We have learned from their experience and include part of their modeling strategies and used their data in CSE-SD, especially in the *Staff Productivity* and *Development Defects and Rework* sectors. The major differences between CSE-SD and the AHM model are summarized as follows:

First, CSE-SD addresses issues that are fundamentally different from those of the AHM. The AHM model addresses software project management issues in general. The model provides a generic software development system dynamics model. CSE-SD is developed to examine the impact of concurrent software engineering on project schedule and effort.

Second, unlike the AHM model, which does not cover the requirements analysis phase, CSE-SD includes five sectors (i.e., *Requirements Work Flow*, *Requirements Defects and Rework*, *Requirements Manpower Allocation*, *Requirements Manpower Needed*, and *Project Scope Change*) to model the requirements analysis phase and address the issues that result from requirements change.

Third, the manpower allocation policy is different. In AHM, manpower resources are allocated to different project-related activities in the order of training, QA, defect rework, then development and testing. In CSE-SD, the remaining daily manpower after training and communication overhead is first distributed to different phases. CSE-SD includes three manpower allocation sectors (*Requirements Manpower Allocation*, *Development Manpower Allocation*, and *SIT Manpower Allocation*); each one is responsible for distributing manpower to different activities within its responsible phase. For example, the *Development Manpower Allocation* sector is responsible for the development (including design and coding) phase. A certain portion of the development manpower is reserved for QA. The remaining manpower is allocated to development defect correction, followed by development activities.

Fourth, CSE-SD breaks down project staff members' daily time into different categories and monitors their changes over time, including project-related time, slack time, training time, intrateam communication time, and interteam communication time. In AHM, training and communication are modeled as a single parameter.

Finally, unlike the AHM model, which is a single-team model, CSE-SD includes the *Interteam Interactions* sector to model the generation, detection, and resolution of problems and issues caused by multiple concurrent teams that could be avoided if done by a single team.

4.4.2 JPL

The Software Engineering and Management Process Simulation (SEPS) model, developed at the Jet Propulsion Laboratory (JPL), is designed to be a planning tool to examine the trade-offs of cost, schedule, and functionality, and to test the implications of different management policies on a project's outcome. The purpose and

model structure of SEPS are similar to those of the AHM model, except that SEPS covers the requirements analysis phase, which is not addressed in the AHM model.

The SEPS model consists of four subsystems: production, staff/effort, scheduling, and budget [50]. The production subsystem models the development progress of a software project. The staff/effort subsystem models the functions which determine the required work force (similar to AHM's human-resource management subsystem). The scheduling subsystem models the functions that determine the time to complete a task and forecasts a completion time for each software life-cycle phase. The budget subsystem keeps track of the cumulative manpower expenditures in relation to available budget.

Like the AHM model, the SEPS model addresses issues that are fundamentally different from those of CSE-SD. The SEPS model addresses software project management issues in general. The model provides a generic software development system dynamics model. CSE-SD has been developed to examine the impact of concurrent software engineering on project schedule and effort.

4.4.3 Madachy

Madachy used the system dynamics approach to study the impact of software inspection on project schedule, effort, and quality [52-53]. The purpose of the Madachy model is fundamentally different from ours. We are interested in assessing the impact of concurrent software engineering on project cost and cycle time. Because the purpose of the model is different, the scope and the formulation of the model therefore are different.

Like the AHM model, the Madachy model covers the design through the system testing phases. However, in the Madachy model, development activities are decomposed into design and coding activities. The main purpose of modeling

design and coding activities independently is to capture the dynamics of defect amplification through successive phases from design through system testing, and to highlight the importance of software inspection in lessening the impacts. Unlike the Madachy model, the CSE-SD model covers the entire software development life cycle, including requirements, development, and system integration and testing.

To examine the impact of CSE, especially the *phase overlapping* concurrency and the *synchronous concurrent subsystems* concurrency, we include four sectors to model the requirements phase (i.e., *Requirements Work Flow*, *Requirements Defects and Rework*, *Requirements Manpower Allocation*, and *Requirements Manpower Needed*), one sector (*Project Scope Change*) to capture the impact of requirements changes, and one sector (*Interteam Interactions*) to address the multiple-team concurrent development issues.

Another difference between the Madachy model and the CSE-SD model, as well as the AHM model, is the QA manpower allocation policy. In the Madachy model, manpower resources are allocated to inspection and rework as needed, as opposed to the Parkinson's manpower allocation policy employed in both the AHM model and the CSE model, where QA is assumed to complete within a certain period, no matter how many tasks are in the queue.

Like the CSE-SD model, the Madachy model assumes that defects are detected only via QA (i.e., inspection) and system testing activities. Project staff members are assumed to be experienced in QA. Unlike CSE-SD, the Madachy model does not consider the effect of project underestimation.

4.4.4 Collofello and Tvedt

Collofello and Tvedt developed a concurrent incremental software development (CISD) system dynamics model in their effort to propose an extensible system

dynamics modeling approach [80]. The CISD model consists of two groups of model components: single-increment components and inter-increment components. The single-increment components group models the development of an increment. It covers the entire software development life cycle, from requirements analysis to system test. The issues and modeling approaches, however, are very similar to those of the AHM model.

The inter-increment component group deserves more attention. It consists of four sectors that deal with inter-increment issues deserve more discussion, namely, *Synchronize Increment Start*, *Increment Overhead Due to Dependence*, *Increment Overhead Due to Overlap*, and *Inter-Increment Defect Regeneration*.

The *Synchronize Increment Start* sector determines when the development of an increment may start. It is determined by the percentage of development and testing completed for every other increment on which this increment depends.

The *Increment Overhead Due to Overlap* sector determines the amount of overhead work of an increment caused by overlapping the development of an increment with all other increments on which it depends. The overhead is decomposed into perceived development overhead, perceived test overhead, underestimated development overhead, and underestimated test overhead. Each category of overhead is modeled as a single parameter. For example, the perceived development overhead incurred by an increment, say Y , due to concurrent development with its dependent increment, say X , is modeled as *init pcvd inc X dev ov oh Y*.

The *Increment Overhead Due to Dependence* sector determines the amount of overhead work of an increment due to its dependence on other increments. Like the *Increment Overhead Due to Dependence* sector, the dependence overhead is decomposed into four categories: perceived development overhead, perceived test overhead, underestimated development overhead, and underestimated test overhead.

For example, the perceived development overhead incurred by an increment, say Y , due to its dependence on another increment, say X , is captured in the *init pcod inc X dev oh Y*. The overhead due to dependence includes modifications, redocumentation, review, and rework of work produced in other increments.

The *Inter-Increment Defect Regeneration* sector determines the defect regeneration of an increment, caused by defect leakage from other increments on which this increment depends. The percentage of defects from a prior increment that will be leaked into this increment is determined based on this increment's relative dependence on other increments. The defects leaked into an increment may be detected by evaluation activities or, by system test, or may leak through system test into the increment's dependent increments.

There are three major differences between CISD and the proposed CSE-SD model. First, CISD is an incremental software development model. It focuses on issues that resulted from overlapping incremental development, such as defect regeneration and overhead incurred by an increment due to reusing any one of its dependent increment's work products.

Second, the modeling approach is different. In their model, each increment is modeled as an instance of single-increment model. The model structure that deals with inter-increment issues needs to be updated every time the number of increments is changed. This modeling approach is not flexible if we want to examine different numbers of increments or if the number of increments is large.

Third, the overhead incurred by an increment due to reusing any one of its dependent increment's work products, modeled as a single generic parameter, is too simplistic. The impact on the client increments due to changes to the reused work by the server increment is not modeled. The overhead incurred by an increment due to overlapping activities has the same problem. We include an "Interteam Interactions"

sector to address the interteam issues that resulted from multiple concurrent activities. This sector models the generation, detection, and resolution of problems and issues caused by multiple concurrent teams that could be avoided if done by a single team.

CHAPTER 5

MODEL TESTING

5.1 Introduction

Before we use the CSE-SD model to assess the impact of concurrent software engineering on project cost and development cycle time, the model has to be tested extensively. Our testing of the CSE-SD model consists of two main steps: unit-level testing and system-level model behavior testing.

The purpose of the unit-level testing is to examine the behavior of each individual model parameter, to make sure each one of them is soundly modeled. In other words, we want to make sure they behave as we expect and they do not produce any anomalous model behaviors. By focusing on each individual parameter and observing their behaviors, we can easily judge the correctness and soundness of the modeling.

CSE-SD is a comprehensive and complicated model which consists of more than 400 model parameters. Therefore, it is impractical to test each one of them individually. Instead, we focus on model parameters that are believed to have significant effects on model behaviors and leave the testing of other parameters to the system-level testing. We perform system-level testing to observe the behavior of the entire model and compare our testing results with those of Abdel-Hamid and Madnick [7] to improve our confidence level of the correctness and soundness of the proposed CSE-SD model. Section 5.2 reports the results of unit-level testing. The results of system-level testing are presented in section 5.3.

5.2 Unit Testing

In this section, we conduct a set of simulation runs to examine the behavior of each individual model parameter to make sure each one of them is soundly modeled. We want to make sure they behave as we expect and do not produce any anomalous model behaviors. By focusing on individual parameters and observing their behaviors, we can easily judge the correctness and soundness of the modeling. We conduct ten test runs to test individual model parameters and sectors. They are described below.

Test Run #1: Perfect Project

Purpose: To test the *Development Work Flow* sector, and the *System Integration and Test* sector. The behaviors of three model parameters are observed: *Cum Units Deved* (cumulative units developed), *Cum Units Integrated* (cumulative units integrated) *Cum Units Tested* (cumulative units tested).

Assumptions:

1. Planned effort equals the actual effort expenditure.
2. No defects are involved.
3. Initial staffing factor is set to 1. That is, 100% of the project's expected staffing is initially allocated.
4. Project staffs spend approximately 50% of their daily time on project-related production activities throughout the entire development life cycle.

Project scenario:

1. The project was accurately estimated to be 64 KLOC large in size (1067 development units).
2. According to the basic organic-mode COCOMO model, 2695 person-days were allocated to the development phase. The planned development production rate, therefore, is $1067/2695 = 0.396$ development units per person-day. The average

staff level is 10. Therefore, 3.96 (0.396 units/person-day \times 10 person-days/day) development units are completed each day.

3. The entire development phase took $1067/3.96 = 270$ working days to complete. As shown in figure 5.1, this is consistent with the result generated from CSE-SD.
4. System integration and test began right after the development phase was completed. As planned (according to COCOMO model), it took 90 working days to complete. To finish system integration and test on time, 0.3 person-days, on average, is needed to integrate and test a development unit. The project took, as estimated, approximately 360 working days to complete.

Conclusions: Under the above assumptions, the model performs as expected.

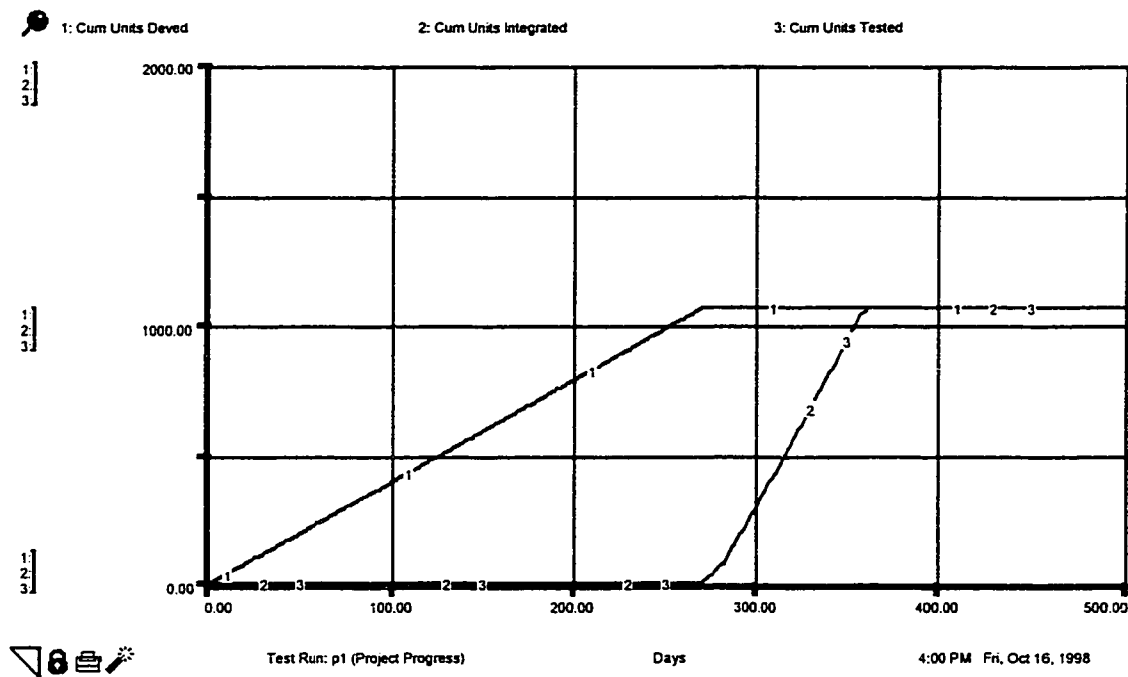


Figure 5.1. Project progress of a perfect project. curve 1: cumulative units developed; curve 2: cumulative units integrated; curve 3: cumulative units tested.

Test Run #2: Effort Underestimation

Purpose: To test the *Project Control* sector. The behaviors of two model parameters are observed: *Planned Project Effort* and *project effort gap reported*.

Assumptions:

1. No defects are involved.
2. Project staffs spend approximately 30% of their daily time on project-related production activities throughout the entire development life cycle.

Project scenario:

There are numerous reasons that cause the manpower perceived still needed to complete the project to deviate from that remaining in the plan. Most significant among the reasons modeled in CSE-SD are:

1. Overestimation of staff productivity: The actual staff productive time is lower than what is planned (i.e., work intensity level is lower than what is assumed in planning).
2. Discovery of unplanned requirements and/or development tasks.
3. Effort underestimation: Planned effort is less than what is actually needed.

When the perceived manpower shortage exceeds a certain threshold, management will adjust the original planned project effort. As shown in figure 5.2, at around day 25, the “project effort gap reported” curve begins to rise. As a consequence, the “Planned Project Effort” curve rises at around day 40. After day 130, the “project effort gap reported” curve begins to drop, indicating a reduced gap between the effort perceived still needed to complete the project and the remaining planned effort. Likewise, the adjustment of “Planned Project Effort” begins to taper off until around day 240. It rises again due to an increased reported project effort gap.

Conclusions: The *Project Control* sector adjusts the planned project effort according to the reported gap between the planned remaining project effort and the perceived effort that is still needed to complete the project.

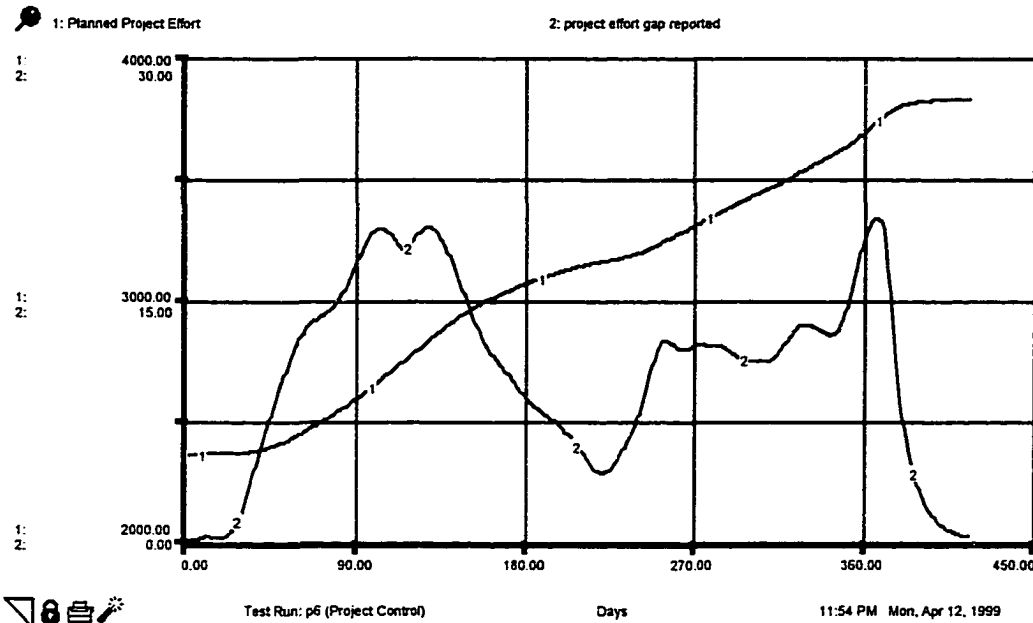


Figure 5.2. Adjusting the planned project effort when there is a reported gap between the perceived project effort needed to complete the project and the remaining project effort.

Test Run # 3: Defects Involved

Purpose: To test the *Development Defects and Rework* sector.

Assumptions:

1. Planned effort equals the actual effort expenditure.
2. The defect densities range in value from 25 defects per KLOC to 12.5 defects per KLOC, with an average value for the project of approximately 19 defects per KLOC [4].

Project scenario:

In CSE-SD, three project factors affecting defect generation rate are modeled. They are defect density, work force mix, and schedule pressure. In this test run, we exclude the impact of all three factors. As illustrated in figure 5.3, when the three factors are not considered, curve 1 (*nominal dev defects per KLOC*) and curve 2 (*dev defects committed per KLOC*) overlap. The impact of each of the three factors is individually tested and are illustrated in figures 5.4, 5.5, and 5.6, respectively.

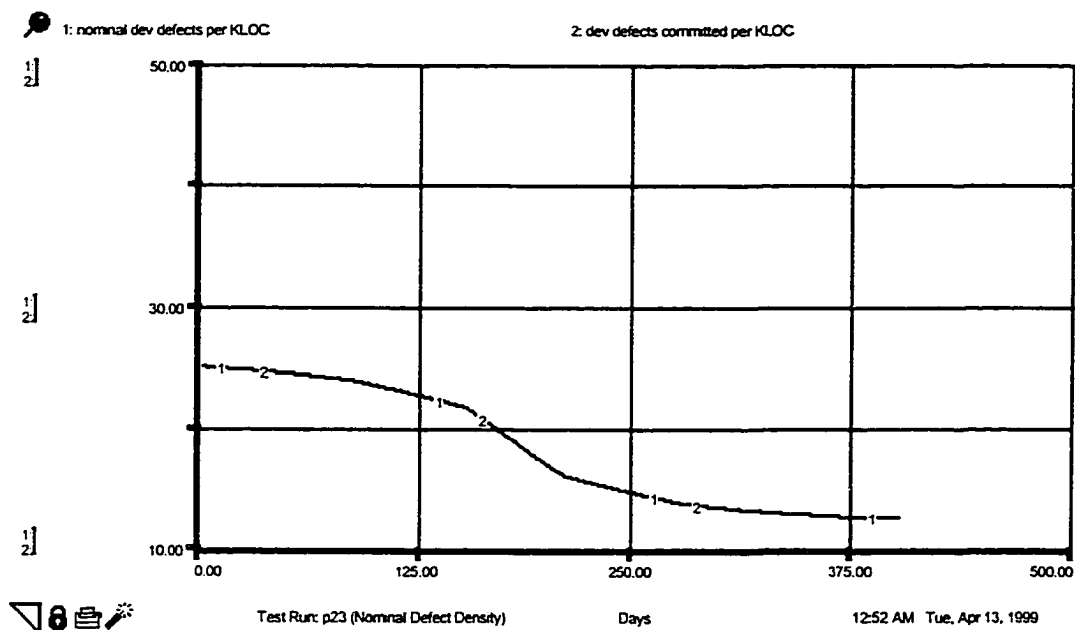


Figure 5.3. Nominal and actual development defect rate.

Test Run # 3.1

Purpose: To test the impact of defect density on development defect generation.

Assumptions: The effects of work force mix and schedule pressure are not included.

Project Scenario:

The rate at which the development defects are generated (*dev defects committed per KLOC*) is determined by multiplying *nominal dev defects per KLOC* (nominal development defects per KLOC) and *dev def density effect on dev def gen* (the development defect density effort on development defect generation).

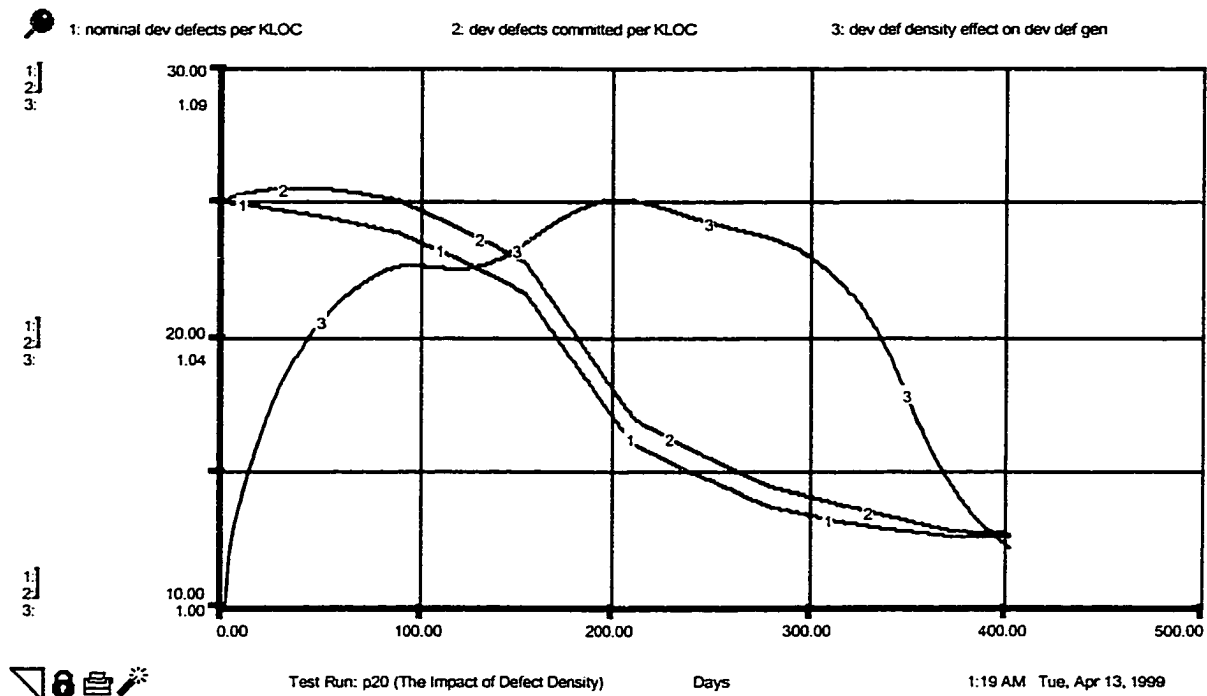


Figure 5.4. The impact of defect density on development defect generation. curve 1: nominal development defects per KLOC; curve 2: development defects committed per KLOC; curve 3: development defect density effect on development defect generation.

Test Run # 3.2

Purpose: To test the impact of workforce mix on development defect generation.

Assumptions:

1. The impacts of defect density and schedule pressure are not included.
2. Initial staffing factor is set to 0.5. That is, only 50% of the project's expected staffing is initially allocated.

Project Scenario:

1. The rate at which the development defects are generated is determined by multiplying development rate and development defects committed per KLOC. That is, *dev def gen rate = dev rate × dev defects committed per KLOC*.
2. As illustrated in figure 5.5, at day 153, curve 5 (*frac staff exp*) begins to drop. This is because management begins to bring in new staff that causes the fraction of experienced staff to drop. As we can see, curve 3 (*dev defects committed per KLOC*) starts to rise and deviate from curve 2 (*nominal dev defects per KLOC*).
3. After a certain period of training and working on the project, new staff members gradually become experienced and more productive. Therefore, curve 5 (*frac staff exp*) rises slowly and reaches 1 at day 388. Curve 2 (*nominal dev defects per KLOC*) and 3 (*dev defects committed per KLOC*) also merge at day 388. This indicates that all new staff members are considered experienced after day 388. Therefore, the impact of workforce mix disappears, since all staff are experienced.

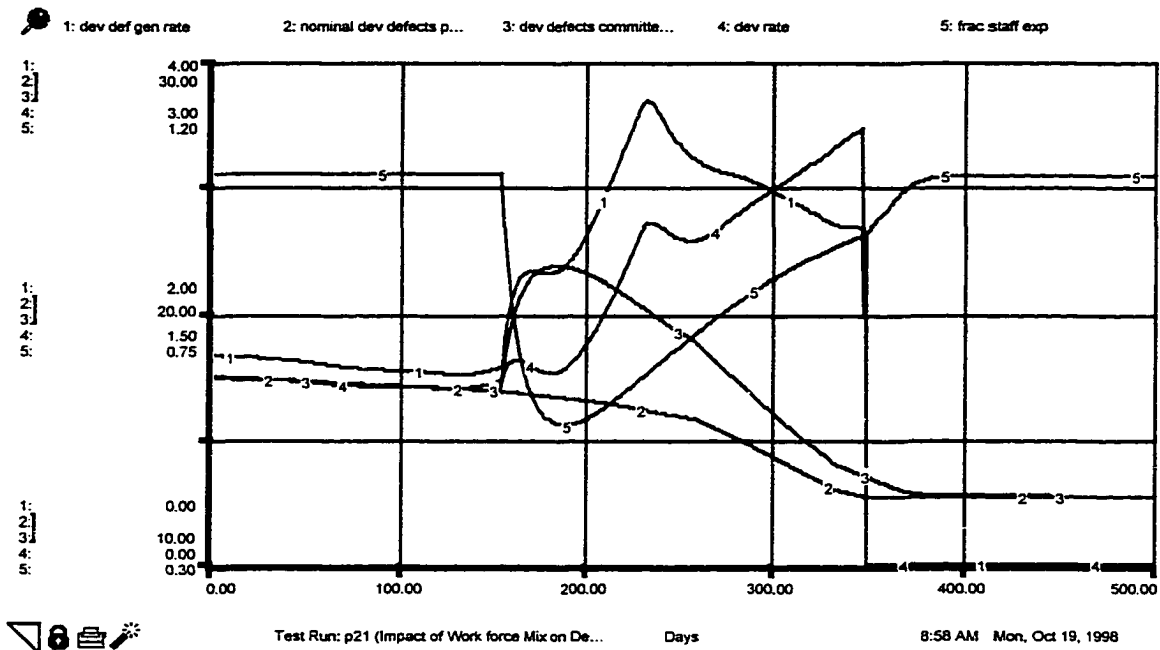


Figure 5.5. The impact of workforce mix on development defects generation. curve 1: development defect generation rate; curve 2: nominal development defects per KLOC; curve 3: development defects committed per KLOC; curve 4: development rate; and curve 5: fraction staff experienced.

Test Run # 3.3

Purpose: To test the impact of schedule pressure on development defect generation.

Assumptions:

1. The impacts of defect density and workforce mix ratio are not included.
2. Initial staffing factor is set to 0.5. That is, only 50% of the project's expected staffing is initially allocated.

Project scenario:

1. As shown in figure 5.6, at around day 50, curve 3 (*schedule pressure*) begins to rise. This is because the perceived manpower still needed to complete the project is

less than the planned manpower that is remaining. As we can see, curve 2 (*dev defects committed per KLOC*) starts to rise and deviate from curve 1 (*nominal dev defects per KLOC*).

2. Management brings in new staff at around day 153.
3. After a certain period of training and assimilation, new staff members become more productive. They gradually close the gap between the perceived manpower still needed to complete the project and the remaining planned manpower. As a result, the schedule pressure is slowly reduced until around day 230.
4. When the schedule pressure is reduced, the gap between *nominal dev defects per KLOC* and *dev defects committed per KLOC* is also reduced. However, after around day 270, even the schedule pressure rises, and the gap between *nominal dev defects per KLOC* and *dev defects committed per KLOC* remains roughly the same. This is because, after day 270, the project already has completed the development phase, and therefore, no development defects are generated.

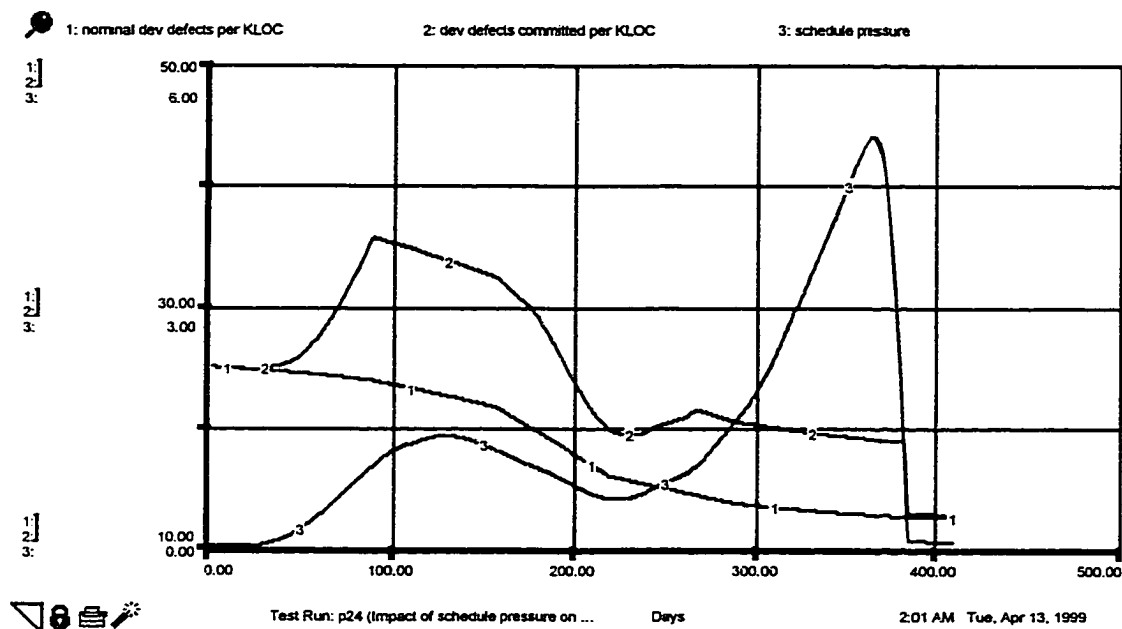


Figure 5.6. The impact of schedule pressure on development defects generation. curve 1: nominal development defects per KLOC; curve 2: development defects committed per KLOC; curve 3: schedule pressure.

Test Run # 4: Project Scope Change

Purpose: To test the *Project Scope Change* sector.

Assumptions:

1. Project staff members spend 60% of their daily time on project-related activities throughout the entire project life cycle. That is, the value of the *daily productive time* parameter is set at 0.6.
2. The project work force level remains unchanged. Therefore, the total daily manpower remains constant throughout the entire project life cycle.

Project scenario:

1. Unplanned requirements are uncovered as the project progress. As shown in figure 5.7, most of the unplanned requirements (85%) are uncovered and incorporated prior to day 200. The maximum number of unplanned requirements uncovered daily is around 0.75 (around day 100).
2. When the unplanned requirements are uncovered, they are incorporated into the project plan. The perception of the project size (curve 2) is increased as a result of the discovery of unplanned requirements.

Conclusions: The *Project Scope Change* sector incorporates requirements changes and updates the perceived project size as expected.

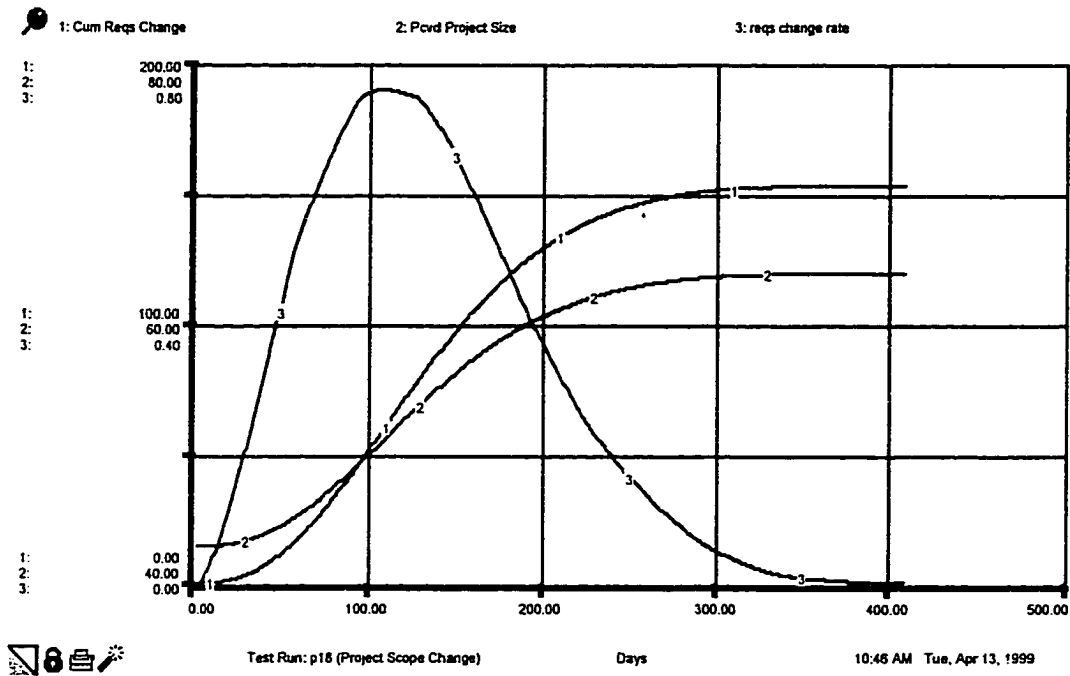


Figure 5.7. Project scope change. curve 1: cumulative requirements change; curve 2: perceived project size (KLOC); curve 3: requirements change rate.

Test Run # 5: Staff Productive Time

Purpose: To test the *Staff Productive Time* sector.

Assumptions:

1. No development defects.
2. No requirements change.
3. Only 50% of the project's expected staffing is initially allocated (i.e., initial staffing factor is set at 0.5).

Project scenario:

1. As shown in figure 5.8, the average training time (curve 1) rises when new staff members (curve 2) are brought into the project. As new staff members are trained and gradually assimilated into the project, they become more experienced. Therefore, the average training time gradually tapers off and approaches 0 at the end of the project.
2. Figure 5.9 depicts the changes in staff members' average slack time (i.e., the time that staff members spend on nonproject-related events each day) and overtime throughout the development project. The factor that drives the changes is the schedule pressure. Schedule pressure occurs when the actual project progress deviates from the planned project progress. An increased schedule pressure then causes staff members to reduce their slack time and, if necessary, to work overtime.

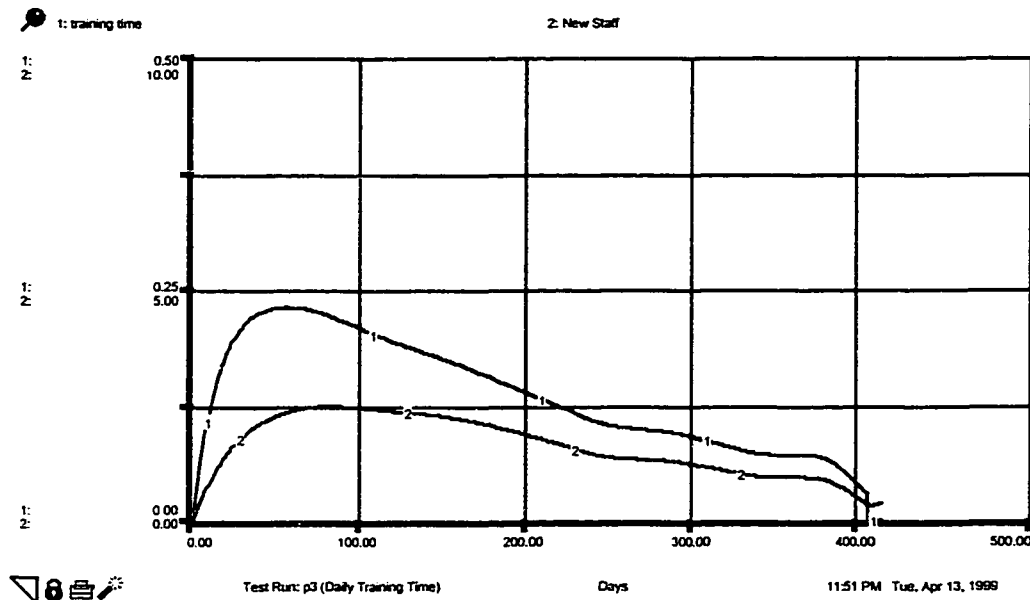


Figure 5.8. Training time.

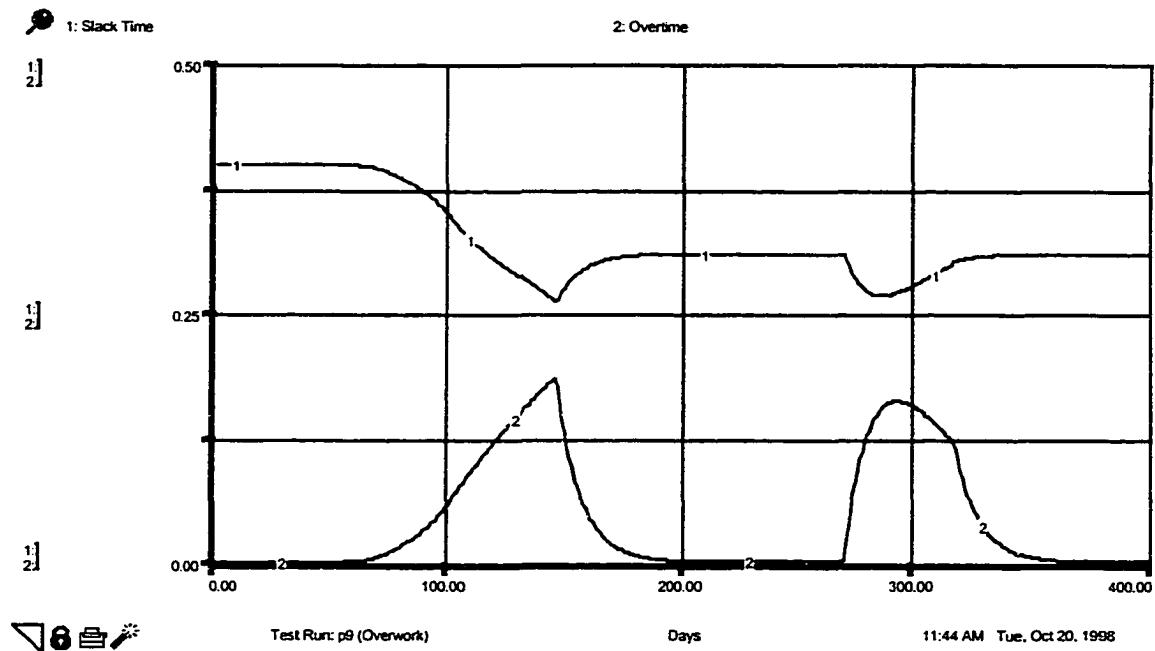


Figure 5.9. Slack time and overtime.

Test Run # 6: Staff Productivity

The purpose of this test run is to test the *Staff Productivity* sector. Specifically, we want to observe how the average staff production rate changes throughout the project. We consider three factors that affect staff members' average production rate in CSE-SD, namely, learning effect, staff exhaustion level, and schedule pressure. Three test runs are performed to test each of the three factors.

Test Run #6.1

Purpose: To test the impact of learning effect on staff productivity.

Assumptions:

1. There are no development defects.
2. No requirements change.

3. Only 50% of the project's expected staffing is initially allocated (i.e., initial staffing factor is set at 0.5).

Project scenario:

1. Figure 5.10 shows the effect of learning on staff production rate. Project staff members will increase their production rate as the project progresses, because they learn while they work on the project.
2. Project staff members will increase their production rate from 60 LOC/person-day in the beginning of the project to 75 LOC/person-day when the project is completed. In other words, project staff members will increase their productivity by 25% through the development of the project.

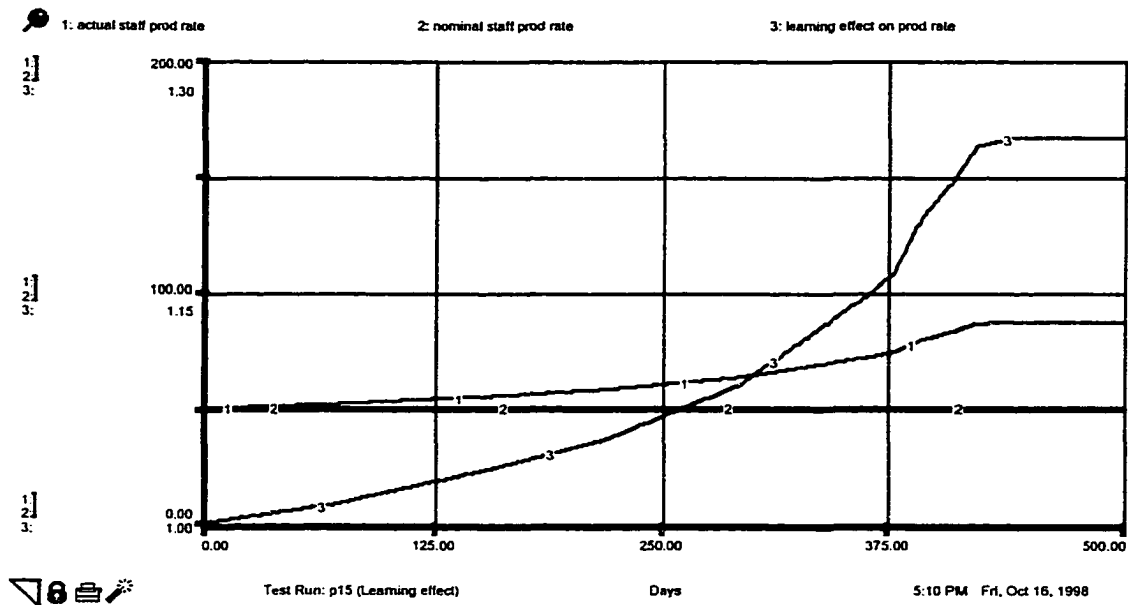


Figure 5.10. Learning effect on staff production rate. curve 1: actual staff production rate; curve 2: nominal staff production rate; curve 3: learning effect on staff production rate.

Test Run #6.2

Purpose: To test the impact of staff exhaustion level on their productivity.

Assumptions:

1. Only 50% of the project's expected staffing is initially allocated (i.e., initial staffing factor is set at 0.5).
2. No requirements change.
3. Nominal staff production rate (*nominal staff prod rate*) is assumed to be a constant (i.e., independent of the workforce mix).

Project scenarios:

As shown in figure 5.11, curve 3 (*Exhaustion Level*) begins to rise when project staff members increase their overwork time (i.e., reduced slack time and/or work overtime). When staff's exhaustion level increases, their production rate will be negatively affected. Curve 1 (*actual staff prod rate*) drops below curve 2 (*nominal staff prod rate*) when staff members' average exhaustion level rises. When their exhaustion level reaches a maximum tolerable threshold, their production rate drops to a minimum, and they are not willing to continue to accept overwork. Without the overwork, their exhaustion level gradually will dismiss. When their exhaustion level

reaches zero (indicating that they are fully recovered from exhausted overwork), they will accept overwork, if needed.

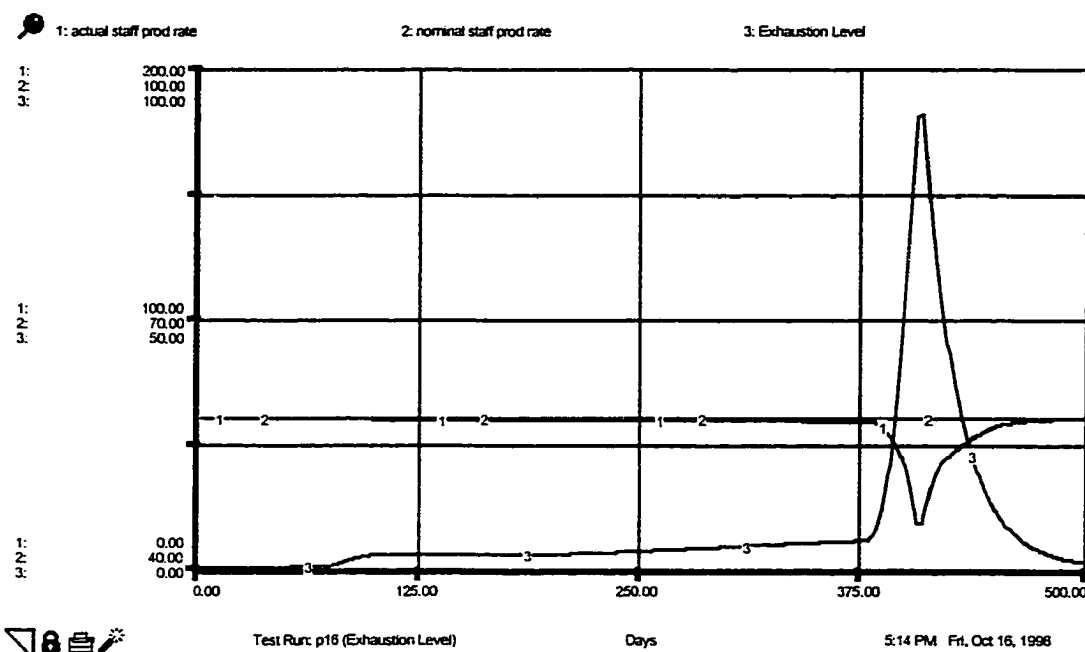


Figure 5.11. The impact of staff exhaustion level on staff production rate. curve 1: actual staff production rate; curve 2: nominal staff production rate; curve 3: exhaustion level.

Test Run # 6.3

Purpose: To test the impact of schedule pressure on staff members' average productivity.

Assumptions:

1. Only 40% of the project's expected staffing initially is allocated (i.e., initial staffing factor is set at 0.4).
2. No requirements change.

3. Nominal staff production rate (*nominal staff prod rate*) is assumed to be a constant (i.e., independent of the workforce mix ratio).

Project scenarios: As shown in figure 5.12, curve 3 (schedule pressure) begins to rise as a result of a perceived gap between the actual development progress and the planned development progress. Curve 2 (actual staff prod rate) also rises in response to the increasing schedule pressure. This indicates that project staff members, when they feel a pressure in their project schedule, will work faster to make up for what has fallen behind.

Conclusions: The data, as depicted in figure 5.12, clearly shows that staff members' average production rate depends on the degree of project schedule pressure.

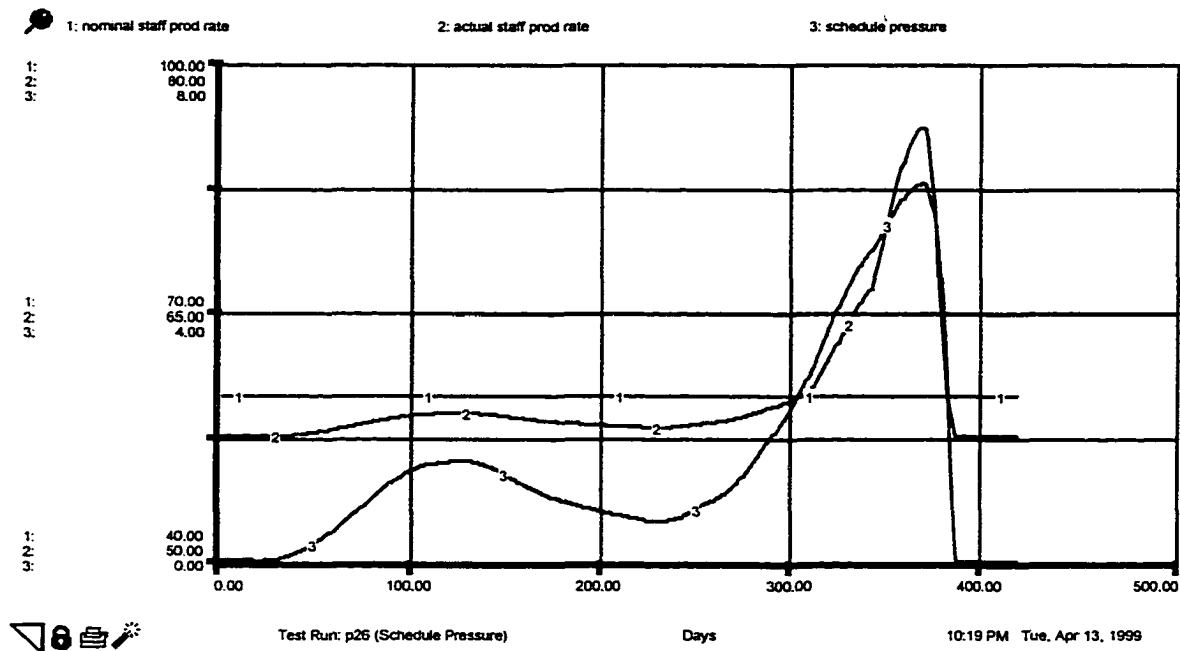


Figure 5.12. The effect of schedule pressure on staff production rate. curve 1: nominal staff production rate; curve 2: actual staff production rate; curve 3: schedule pressure.

5.3 System Testing

To place faith in simulation model-based analyses and policy recommendations, we have to know the degree to which those analyses might change as reasonable alternative assumptions are built into the model. First, we want to make sure our model produces similar behavior, with minor variations in equation formulations and parameter values. Next, we want to know if the CSE-SD model is capable of generating project behaviors similar to those reported in the literature. To conduct the test, we calibrate CSE-SD against the data reported in Abdel-Hamid and Madnick [7]. Our purpose is twofold: (1) to use their data and simulated results as a reference and (2) to compare our simulated results with theirs. The key statistics of the

project (called EXAMPLE) that we compare are summarized in appendix C. More detailed information is included in [7].

Since the requirements phase and multiple-team concurrent development issues are not addressed in the AHM model, data is not completely available to fully validate the entire model. The model components that are validated include *Project Control*, *Development Manpower Needed*, *System Integration and Test Manpower Needed*, *Development Defects and Rework*, *Work Force*, *Staff Productive Time*, *Staff Productivity*, *Development Manpower Allocation*, *System Integration and Test Manpower Allocation*, *Development Work Flow*, *System Integration and Test*, *Project Scope Change*, and *Planning*. Model components that are not validated include *Interteam Interactions*, *Requirements Manpower Allocation*, and *Requirements Manpower Needed*. These three sectors are calibrated against the COCOMO model [22-23, 29] and are tested in chapter 7.

We compare seven key project measures, namely, perceived job size, perceived project cost, cumulative units developed, cumulative units tested, scheduled completion date, cumulative project cost, and work force distribution pattern. The comparisons of these key project statistics are illustrated in figures 5.13 to 5.16. Figure 5.13 displays three key project measures: perceived project size, cumulative units developed, and cumulative units tested. The “perceived project size” curve depicts the pattern of how the project scope was changed over time after the discovery of unplanned development units. The real size of the project is 64 KLOC (1067 development units), but was initially estimated to be 42.88 KLOC (715 development units). The “cumulative units developed” and “cumulative units tested” curves show how the development units are completed and tested over time. The patterns of these two curves are very close to that of the AHM, especially in the first half (prior to day 220) of the “cumulative units developed” curve. However, after that, the CSE-SD simulated project progress is faster than that of the AHM, although the difference is not

significant. The reason that causes the difference stems from the difference in work force level. As indicated in figure 5.16, after day 220, CSE-SD has a higher work force level than that of the AHM. More work force means more tasks can be done within the same period of time.

Figure 5.14 shows the cumulative project effort expenditure and the change in estimated project cost. Our simulated "cumulative project cost" curve is almost identical to that of the AHM prior to day 200. AHM produces a higher effort expenditure after day 200 because its work force curve reaches the peak earlier than that of CSE-SD. In fact, the AHM simulated project has more people on board than that of CSE-SD within the period of day 160 to 220. This explains why the project cost accumulates at a faster pace in AHM than in CSE-SD.

The "perceived project cost" curve shows how management adjusts the estimated project cost as a result of the discovery of unplanned development units. Overall, the two simulated curves are similar before day 280. After that, the AHM curve displays an immediate uprise which is not seen in the CSE-SD curve. The reason for the difference lies in the difference in project control mechanisms. In CSE-SD, when new tasks are discovered, the adjustment of estimated project costs includes both the development cost and the system testing cost. The estimated project cost is adjusted well before conducting the system test phase. Therefore, we do not see any sharp change in the perception of the project cost right before conducting system test. However, the AHM model adjusts the estimated project cost right before and during the system test phase. The other reason that causes the difference is that CSE-SD has a smaller project cost than that of the AHM (3620 man-days in CSE-SD as opposed to 3795 man-days in AHM).

As illustrated in figure 5.16, CSE-SD produces a Rayleigh-curve work force distribution pattern, with a peak work force of around 11 project staff. The shape of the

curve is very close to that of the AHM model. However, after around day 220, the CSE-SD curve deviates from that of the AHM. The CSE-SD work force curve reaches its peak at around day 220 and gradually tapers off to around 8 staff on board at the end of the project. However, in AHM, the work force curve reaches its peak at around day 190 and gradually tapers off to around 7 staff on board at the end of the project.

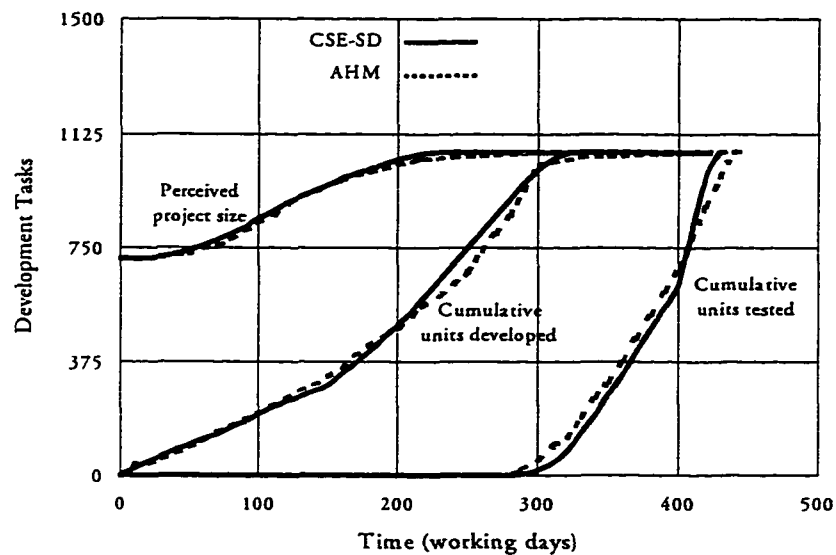


Figure 5.13. Comparison of project progress of the EXAMPLE project.

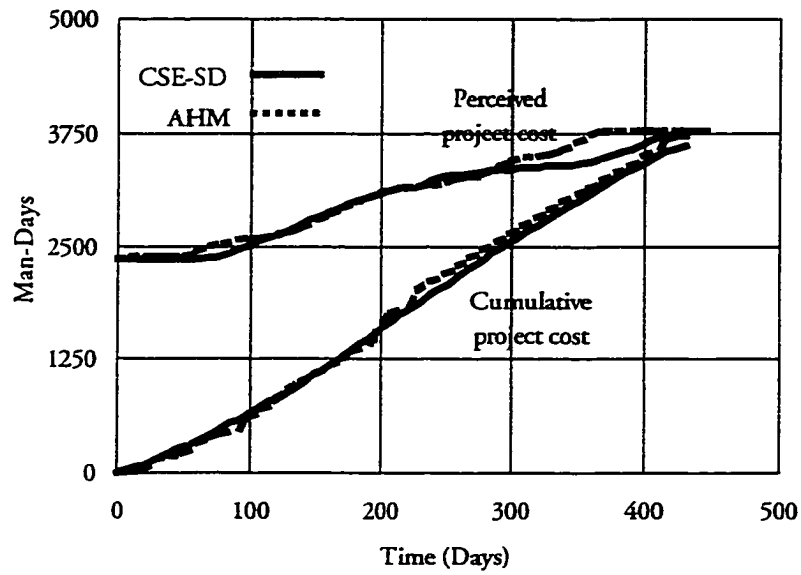


Figure 5.14. Comparison of project cost of the EXAMPLE project.

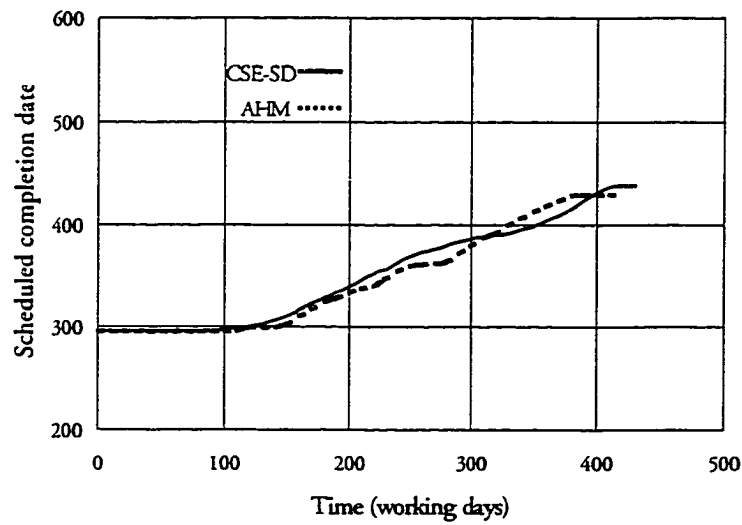


Figure 5.15. Comparison of scheduled completion date of the EXAMPLE project.

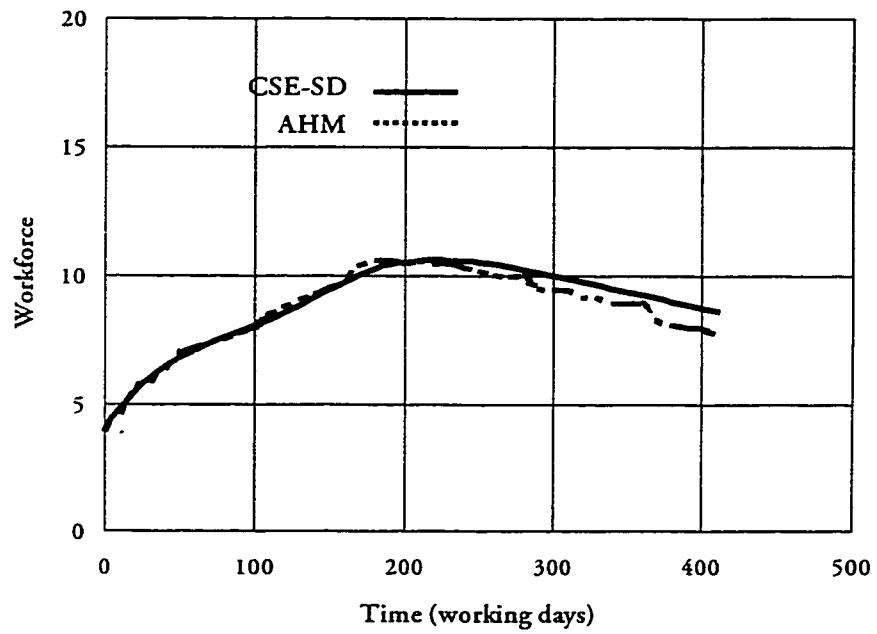


Figure 5.16. Comparison of work force distribution of the EXAMPLE project.

CHAPTER 6

BROOKS' LAW REVISITED

6.1 Introduction

Despite the recent advances in software development and management technologies, software development continues to suffer schedule delays and budget overruns. When a project is behind schedule, software managers respond by bringing people into the project. The result is, as suggested by the famous Brooks' Law [24], a further delayed or even collapsed project. Brooks developed the law through observation of many projects and derived the generalization. His explanation was quite reasonable and convincing. However, it becomes a debilitating statement to any software project manager who is faced with a late project.

In this chapter, we perform an in-depth study using the proposed CSE-SD model. In specific, we will use CSE-SD to answer two questions: (1) What is the impact of adding people late in a software project? Will the project be completed earlier or be delayed even further, as predicted by Brooks' Law? and (2) When is the best time to add people into a software project and how many people should be added?

The remainder of this chapter is organized as follows. Related studies on Brooks' Law are reviewed in section 6.2. Section 6.3 examines the dynamic implications of Brooks' Law. The results of our study are presented in section .

6.2 Related Studies on Brooks' Law

Brooks' Law has been addressed extensively in the past. Gordon and Lamb studied Brooks' Law and suggested that the best way to recover from a slipping

schedule is to add more people than might be expected to be necessary, and to add them early [38]. Three factors are considered in their study: time loss due to new staff learning, time loss due to teaching by experienced staff, and time loss due to group communication. They suggest adding more manpower than you think is necessary as soon as you sense trouble, then do not change anybody's job until the project is finished.

Weinberg addresses Brooks' Law from the system dynamics perspective [84]. He argues that the effect of Brooks' Law is caused by an increased coordination and training overhead. More coordination overhead means more work has to be done. The increased training load on the experienced workers leads to a reduced amount of productive work being done. The effect of Brooks' Law can be made even worse when management takes erroneous actions. For example, when management waits too long to communicate the problem and attempts "big" corrective actions, this usually leads to a project collapse.

Abdel-Hamid and Madnick studied Brooks' Law using their system dynamics model. Two important, but unrealistic, assumptions are made in their study. First, their model assumed that development tasks can be partitioned, but that there is no sequential constraint among them. The development production rate depends solely on available manpower, not on sequential constraint. In reality, if tasks have to be done sequentially, then adding more people will not speed up the development process, since there are not enough tasks ready for them to work on. You expend people hours, but get little results [64]. The number of months of a project depends upon its sequential constraints. The maximum number of staff members depends upon the number of independent subtasks.

Another assumption is that project managers continuously will add new people as long as they sense a shortage in manpower. In reality, project managers can

only add new people a few times throughout the entire project life cycle. The two unrealistic assumptions lead to their conclusion that “adding more people to a late project always causes it to become more costly but does not always cause it to be completed later” ([4], [7]). The increase in the cost of the project is caused by the increased training and communication overhead, which, in effect, decreases the productivity of the average team member and, thus, increases the project’s person-day requirements. Only when the incurred training and communication overheads outweigh the increased productive manpower will the addition of new staff members translate into a later project completion time.

In this chapter, we study Brooks’ Law using more realistic assumptions. The sequential constraint of a software project is considered in our model. We also make an assumption that people are added into the project only once throughout the entire development life cycle, because it is not easy to obtain approvals from upper management to add manpower frequently to any project.

6.3 Dynamics of Brooks’ Law

The dynamics of Brooks’ Law starts with management bringing new staff into a project. Three effects, as illustrated in figure 6.1, are: (1) an increase in communication and training overhead, (2) an increase in the amount of work repartitioning, and (3) an increase in the total manpower available for project development.

When new staff are brought in, they require a certain level of training, and this will take away part of the old staff’s productive time. Also, more people require more communication. As a result, the total project manpower resources also decrease. Less total project manpower means less manpower for development and decreased average work rate. This results in project progress being delayed even further and leads to another round in the people-hiring feedback loop.

The second effect of bringing in new people midway in the project occurs when work needs to be repartitioned. The work currently being performed by old staff needs to be repartitioned so some of it can be assigned to new staff. Project staff, both new and old, have to adapt to, and learn, new tasks. The coordination overhead also is increased, especially when the work is not well partitioned.

Another impact of bringing in new people is that more people are available to be assigned to the project. As a result, the average work rate, as determined by the total number of project staff and the average staff productivity, also increases. An increase in the average work rate means work is being done at a faster pace, and eventually will catch up with planned progress. As a result, the degree of schedule slippage is reduced, which reduces the need to bring new people into the project.

As schedule pressure rises, part of the planned QA work might be skipped. As a result, the defects contained within the work product remain undetected, which leads to defect amplification. Also, under extreme schedule pressure, project staff are prone to commit more defects than normal. The increased amount of defects means that part of the planned manpower for development now has to be devoted to defect correction. With less manpower available for development, the project is delayed even further, which causes the schedule pressure to rise and triggers another defect amplification “vicious cycle” [9].

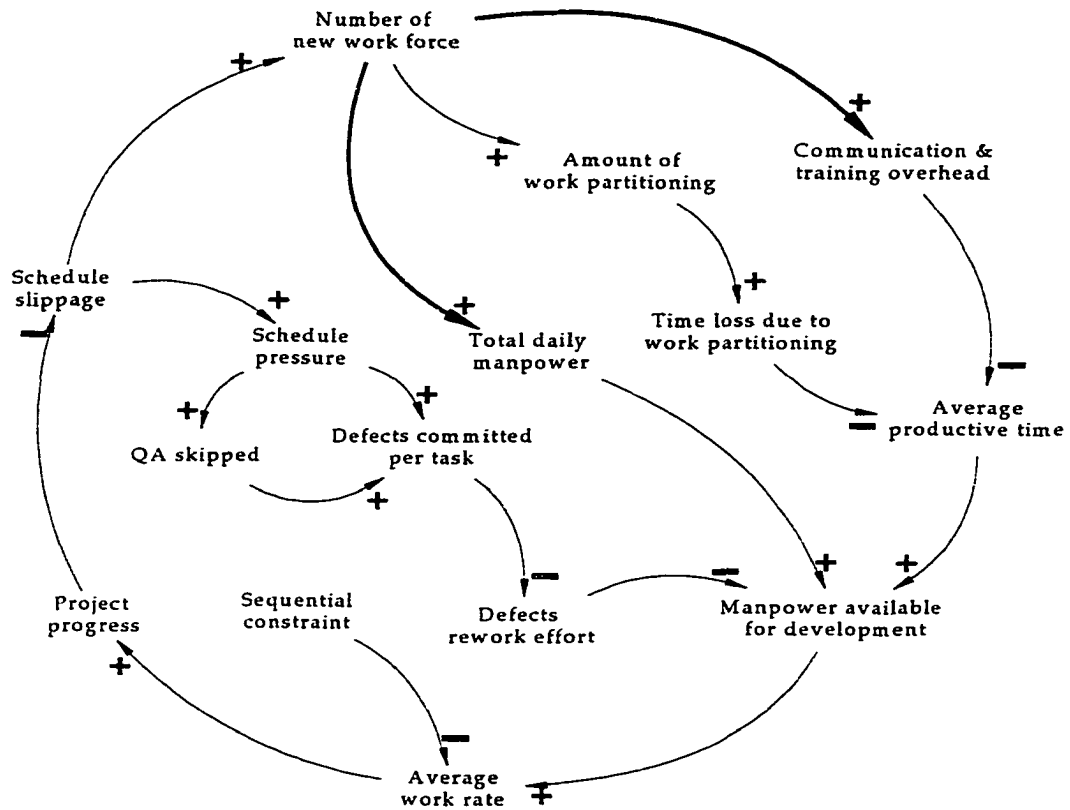


Figure 6.1. The dynamics of Brooks' Law.

Unlike the Abdel-Hamid and Madnick model, we take the sequential constraint of a software project into consideration in our model. One simple approach to model sequential constraint is to sample a software development PERT chart into a sequence of task groups $\langle TG_1, TG_2, TG_3, \dots, TG_n \rangle$. Tasks within TG_2 have to wait for all the tasks in TG_1 to finish before they can start. When all tasks in TG_1 are completed, the project is perceived to be N_1/N completed, where N_1 and N are the total number of tasks in TG_1 and in the entire project, respectively. For example, as shown in figure 6.4 (a), we sample the PERT chart into a sequence of four task groups,

namely, $\langle \{Requirements, Test Plan\}, \{Design, Test Data, Test Drivers\}, \{Code, Document\}, \{Product Test\} \rangle$. The *Design* task, which is in the second task group, has to wait for the *Requirements* task to complete before it can start. When all the tasks within the first task group are completed, the project is perceived to be 25% (i.e., 2/8) complete. The project proceeds to the second task groups with 37.5% of the tasks ready for assignment. They can be performed at the same time and in any order.

Sequential constraint, as modeled as “degree of concurrency” (DC), is defined as the fraction of the number of tasks (including development and testing) that are ready to be worked on and the number of tasks project staff are able to perform. As shown in figure 6.4 (b), the number of tasks that project staff can perform is determined by multiplying “the amount of daily manpower allocated” by “staff’s average productivity.” For example, degree of concurrency = 0.8 means only 80% of the tasks that project staff are able to perform are ready for assignment. To simplify, we assume that the discovered unplanned tasks are uniformly distributed among task intervals. Therefore, the degree of concurrency remains unchanged before and after unplanned tasks are discovered. By changing the values of this parameter, we can examine the impact of different degrees of sequential constraint on project cost and development cycle time.

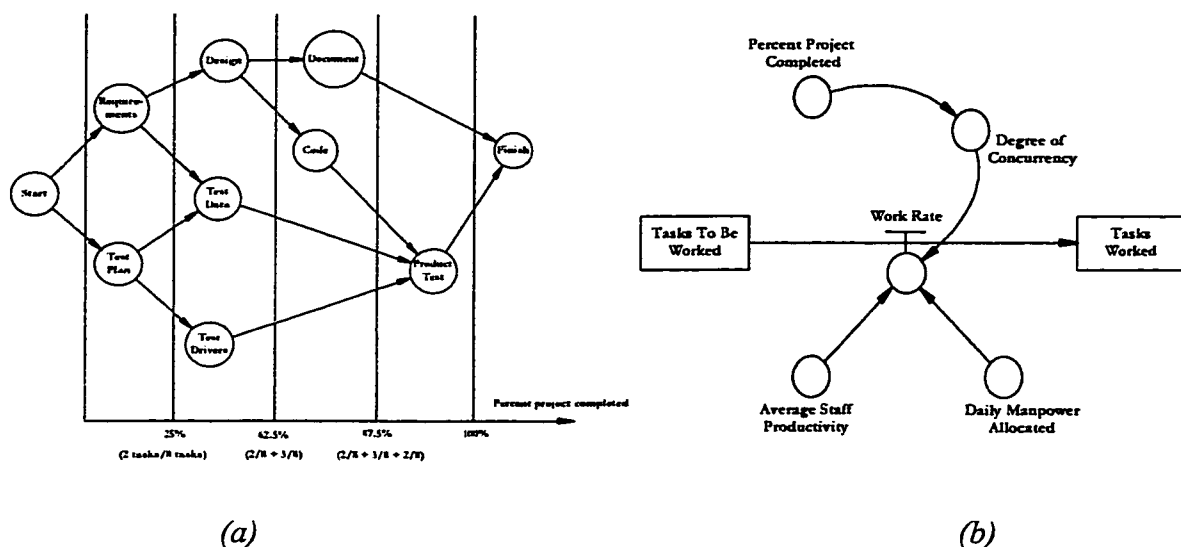


Figure 6.2. Modeling sequential constraint. (a) A simple software development PERT chart [22]; (b) The development and testing rate depend on sequential constraint.

6.4 Simulation Results

We present our simulation results with a focus on two questions: (1) What is the impact of adding people to a software project, in terms of project completion time and cost? And, (2) When is the best time to add people into a software project, and how many people should be added? We first address question 1: what is the impact of adding people to a software project, in terms of project completion time and cost? To answer this question and to compare our results with those of Abdel-Hamid and Madnick (AHM), we use, in this study, the same manpower addition assumption that they did. However, we add the sequential constraint factor to reveal its effect. We continue to add people as long as there is a shortage in manpower until a preset date. For example, as indicated in figure 6.3 (a), for a project without

sequential constraint (i.e., $DC = 1.0$), if we continue to add people whenever we sense a shortage in manpower until 36 (i.e., $0.3 \cdot 120$) working days remaining in the planned project schedule, then the project is expected to complete at around day 435. However, after 180 working days remain (i.e., $1.5 \cdot 120$), management is not 100% willing to hire enough people as desired. The total cost of the project is 3686 person-days, as shown in figure 6.3 (b).

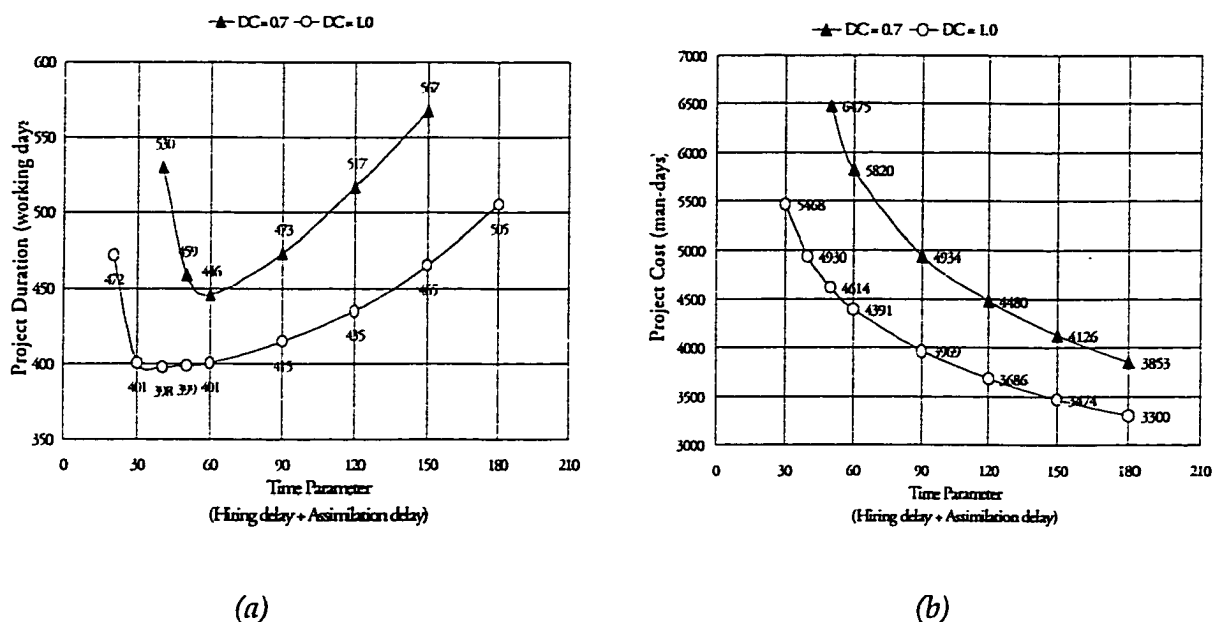


Figure 6.3. The impact of work force stability on project duration and cost. (a) project duration; (b) project cost.

As shown in figure 6.3, a more aggressive manpower acquisition policy results in a shorter project duration, but increases project cost. We simulated different manpower acquisition policies by changing the value of Time Parameter (TP) and determined that 398 working days is the shortest possible schedule one can achieve for this specific project. Time Parameter is defined as the sum of the time to hire new

staff (hiring delay) and the time to train and assimilate new hires (assimilation delay). Our results indicate that Brooks' Law holds only when the Time Parameter is fewer than 40 working days for a medium-sized COCOMO organic-mode project EXAMPLE. Our result is very similar to that of AHM.

The trend for the $DC = 0.7$ project (i.e., a project with a certain degree of sequential constraint) is similar to that of $DC = 1.0$ project; project duration continues to decrease when new work force is added. However, management pays the price of increasing project cost. For projects with certain degree of sequential constraint (i.e., $DC = 0.7$), Brooks' Law holds when the Time Parameter is less than 60 working days—about one month earlier than that of the $DC = 1.0$ project (40 working days). This implies that sequential constraint does play a role in this situation. If management fails to sense the shortage in manpower and does not make a timely decision to add work force, then the project will be delayed further, especially if there is a certain degree of sequential constraint among development tasks. Project cost continues to rise when new people are added, as illustrated in figure 6.3 (b). Project cost increases nonlinearly when Time Parameter is less than 90 working days. This implies that adopting a more aggressive manpower acquisition policy late in the project will cost more.

Figure 6.4 shows the impact of sequential constraint on project duration and cost. As expected, as the degree of sequential constraint increases (degree of concurrency decreases), project duration will increase, and so does the project cost. However, project duration and cost increase nonlinearly when DC is less than 0.5. The result indicates that a tighter sequential constraint has a stronger negative impact on project duration and cost.

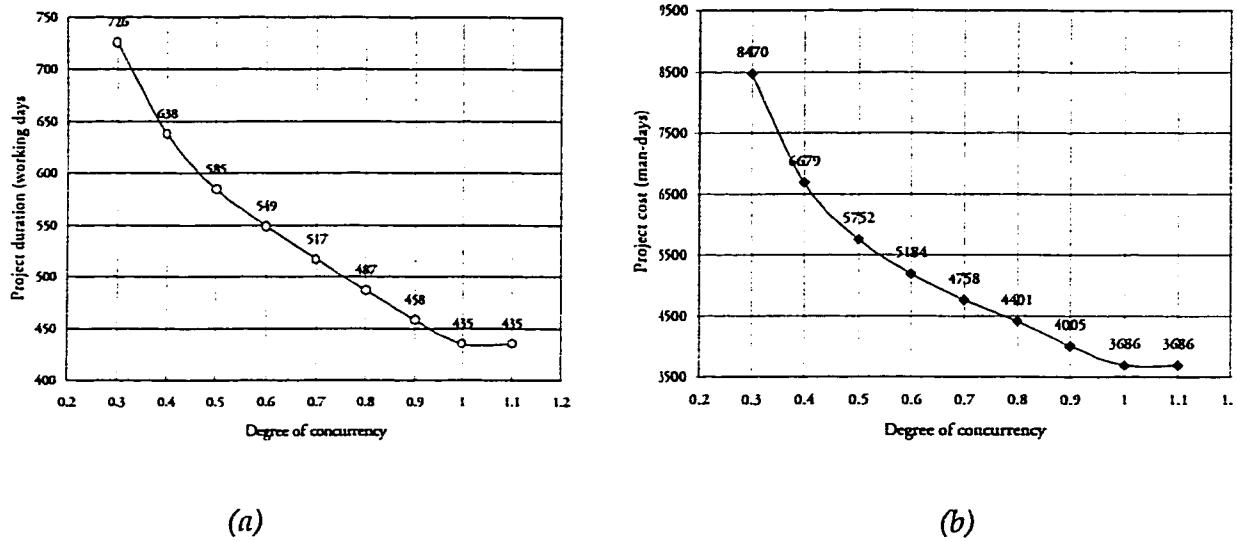


Figure 6.4. The impact of degree of concurrency and project duration and cost. (a) project duration; (b) project cost.

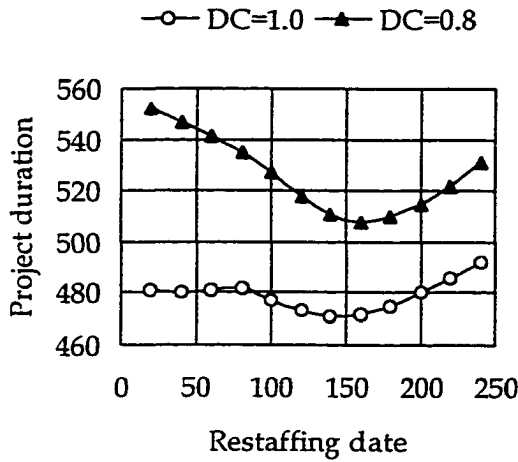
We next address question 2: what is the best time to add people into a software project and how many people should be added? Unlike the AHM model, we take the sequential constraint of a software project into consideration. Besides, to answer the question, we make a more realistic assumption that people are added into the project only once throughout the entire development life cycle. We conducted 24 simulation runs; 12 on projects with perfectly partitionable tasks (*PPT*) [24] and 12 on projects with a certain degree of sequential constraint. The results are summarized in figure 6.5.

At the specified milestone date, the desired work force that is needed to complete the project on time is brought into the project. For example, at day 20 (one month after the project was launched), the desired new work force perceived needed to complete the project on time is 4.16 for *PPT* projects (i.e., $DC = 1.0$). This is

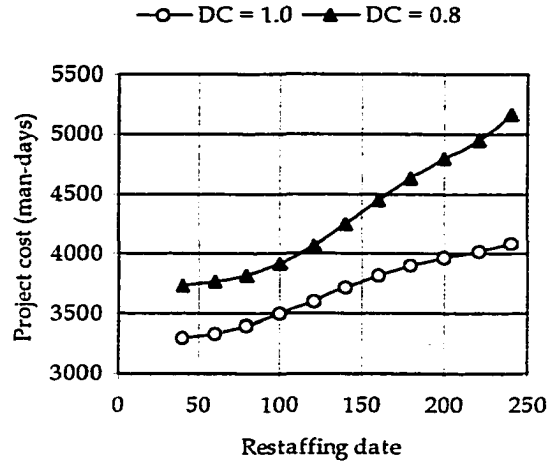
because, in the beginning of the project, there are only four engineers on board, while eight engineers are expected (i.e., 50% understaffed). Management will need to bring in the other four people as initially planned plus extra manpower to make up for the delayed work caused by having four engineers doing the work that is expected by eight engineers. In this specific organic-mode project, there is a threshold time T —about one-third (i.e., $140/472$) of the development life cycle—before which adding people into a software project will not extend project duration. However, after the threshold time, adding people to the project will cause the scheduled completion date to extend.

The $DC = 0.8$ project also has a threshold time at day 160; one month (20 working days) later than that of the $DC = 1.0$ project. However, it is also at about one-third (i.e., $160/508$) of the entire development life cycle. After simulating projects with different degrees of concurrency (from $DC = 0.5$ to 1.0), we found that the threshold time will shift forward as the degree of concurrency increases. But the one-third point does not change. As shown in figure 6.5 (b), adding people into a software project will, in general, cause the project cost to increase.

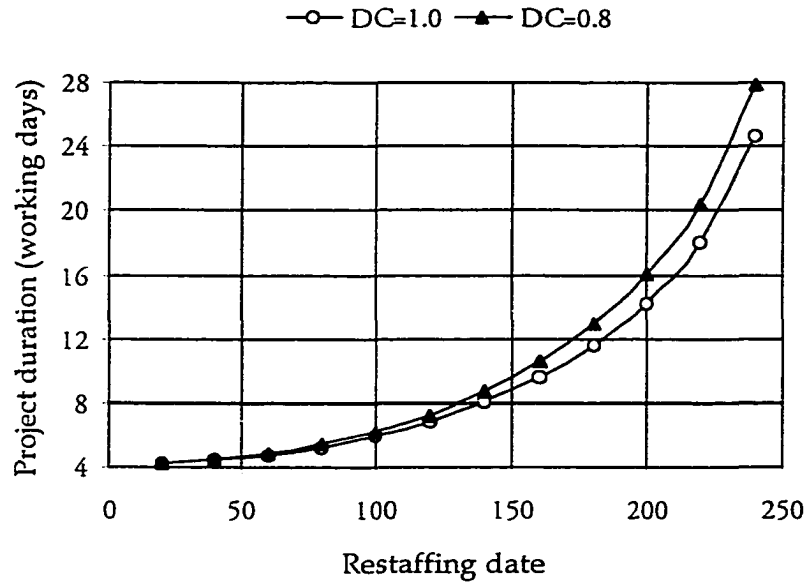
There could be numerous alternatives between the two extreme manpower acquisition policies we use in our simulation runs, namely, continuous manpower acquisition policy and one-time manpower acquisition policy. The outcomes of adopting different manpower acquisition policies are expected to fall between our simulated results. For organic-mode projects, we predict a project schedule-effective time range from one-third of the project to halfway into the project life cycle.



(a)



(b)



(c)

Figure 6.5. Impact of restaffing time on project duration, cost, and number of needed work force. (a) project duration; (b) project cost; (c) number of needed new work force.

6.5 Summary

We performed an in-depth study of Brooks' Law using the CSE-SD model. The results of the study are based on three sets of simulation runs with different assumptions. First, we used the same assumptions as those of AHM: (1) project tasks can be partitioned, but there is no sequential constraint among them; and (2) management continuously will add new people as long as they sense a shortage in manpower. Under these assumptions, our results are consistent with those of AHM, namely, adding more people to a late project always causes it to become more costly but does not always cause it to be completed later.

Next, we used a more realistic assumption by considering sequential constraint. We found out that continuously adding people to a late project makes it later and more costly. This confirms Brooks' Law. However, these results are not consistent with those of AHM's. This implies that sequential constraint does play a role in project development.

Finally, we added another realistic assumption that people are added to a project only once throughout the entire project life cycle because it is difficult to obtain frequent manpower addition approvals from upper management. We found that there is an optimal time range for adding people without delaying a project. It ranges from one-third to halfway into the project development. If software project managers cannot make a timely and accurate decision on project restaffing prior to halfway into the project, the project has a high probability of being delayed, especially when task sequential constraints are involved. However, adding people during the project always causes the project cost to increase.

In summary, it is always costly to add people to a late project. When sequential constraint is significant, adding people late in a project will make it later. We also

have found, in this study, an optimal time range for adding people without delaying a project.

CHAPTER 7

ON THE IMPACT OF CONCURRENT SOFTWARE ENGINEERING

7.1 Introduction

In this chapter, we conduct a set of simulation experiments using the CSE-SD model. The objective of the experimentation is twofold: (1) to further demonstrate the capability of CSE-SD to serve as a management policy exploration tool; and (2) to investigate the impact of concurrent software engineering on project cost and development cycle time. Specifically, the following two sets of questions are addressed:

1. What are the effects of the “phase overlapping” development approach on project cost and development cycle time? Will phase overlapping reduce project duration and/or cost? What is the optimal degree of phase overlapping in terms of project cost and development cycle time? In other words, what are the best degrees of phase overlapping that lead to shortest project duration and/or lowest project cost?
2. What are the effects of the “synchronous concurrent subsystems (SCS)” development approach on project duration and cost? Will the SCS approach reduce project duration and/or cost? For a given project, what is the optimal number of subsystems (subteams) that lead to the shortest project duration and lowest cost?

Before we use the CSE-SD model to answer the above questions, we need to select appropriate values for model parameters. To assess the impact of concurrent software engineering practice in general, we will calibrate CSE-SD against the

COCOMO model. Calibration refers to assigning specific values to model parameters that produce project behaviors similar to those predicted by COCOMO.

Model calibration is presented in section 7.2. The first set of questions regarding the effects of the “Phase Overlapping” development approach is addressed in section 7.3. The effects of the “Synchronous Concurrent Subsystems” development approach are investigated in section 7.4 where the second set of questions are addressed.

7.2 Model Calibration

To examine the effects of the Phase Overlapping concurrent development approach, we calibrate CSE-SD against the COCOMO 2.0 model [23].

7.2.1 The BASELINE Software Project

We use a baseline COCOMO 2.0 (called BASELINE) project as a reference to examine the effects of the Phase Overlapping development approach using the CSE-SD model. BASELINE is a 128 KLOC large project with the values of the seventeen COCOMO 2.0 cost drivers and five scale factors are set to “nominal.” In COCOMO, the software development process is divided into four major phases: Plan and Requirements, Product Design, Programming, and Integration and Test. The overall phase distribution of project effort, schedule, and full-time-equivalent software personnel (FSWP) for the BASELINE project is summarized in table 7.1.

COCOMO-estimated project development effort, including the effort spent in the Plan and Requirements phase, the Product Design phase, the Programming phase, and the Integration and Test phase, is 704.9 person-months (i.e., $46.0 + 112.0 + 362.4 + 184.5$), or 13,393 person-days (PDs).

Table 7.1. Phase distribution of project effort, schedule and personnel

PHASE	EFFORT (person-months)	SCHEDULE (month)	FSWP
Plans and Requirements	46.1 (876 person-days)	6.0	7.7
Product Design	112.0 (2128 person-days)	7.4	15.2
Programming	362.4 (6886 person-days)	12.0	30.2
- Detailed Design	158.1	-	-
- Code and Unit Test	204.2	-	-
Integration and Test	184.5 (3506 person-days)	7.9	23.3

7.2.2 Mapping COCOMO Development Activities to CSE-SD

COCOMO includes eight major activities: *requirements analysis*, *product design*, *programming*, *test planning*, *verification and validation*, *CM/QA*, *project office functions*, and *manuals* [22]. The *requirements analysis* activity is modeled in the *Requirements Work Flow* sector. The *product design* and *programming* activities are mapped to the *Development Work Flow* sector. The *verification and validation* activity performed during the *Integration and Test* phase is modeled in the *System Integration and Test* sector.

The QA activity modeled in CSE-SD includes the *verification and validation* and *CM/QA* activities performed in COCOMO's *Plans and Requirements*, *Product Design*, and *Programming* phases. The *test planning*, *project office*, and *manuals* activities are considered part of the requirements specification and software development in CSE-SD. For example, during the requirements phase, the *manuals* activity dealing with outlining portions of users' manual is considered part of a requirements specification activity.

The distributions of project effort, schedule, and personnel of the eight different COCOMO activities in each phase are shown in tables 7.2, 7.3, 7.4, and 7.5, respectively. For example, table 7.3 shows the breakdown of project effort, schedule, and personnel in the "Product Design" phase. In the "Product Design" phase, 14.0

person-months of effort are spent on the *requirements analysis* activity, 45.9 person-months of effort are spent on the *product design* activity, 15.1 person-months of effort are spent on the *programming* activity, and so on.

Based on the above COCOMO-to-CSE-SD mapping and the data summarized in tables 7.2 to 7.5, equivalent CSE-SD distributions of project effort and schedule are summarized in tables 7.6 to 7.11. The data listed in table 7.6 show the effort, schedule, and personnel distribution of the BASELINE project without requirements change. The data are derived by setting the COCOMO 2.0 BRAK (breakage percentage) factor to 0%. They are calculated as follows:

- Requirements phase
 1. Requirements specification effort = $(20.8 + 1.8 + 5.8 + 2.3)$ person-months * 19 working day/month = 583 person-days
 2. Requirements QA effort = $(3.5 + 1.4)$ person-months * 19 working days/month = 93 person-days
 3. Development effort spent in the Requirements phase = $(8.1 + 2.5)$ person-months * 19 working days/month = 201 person-days
- Development phase
 1. Software development effort = $((45.9 + 15.1 + 6.7 + 11.2 + 7.8) + (29.0 + 204.7 + 19.9 + 21.7 + 18.1))$ person-months * 19 working day/month = 7,222 person-days
 2. Development QA effort = $((8.4 + 2.8) + (30.8 + 23.6))$ person-months * 19 working days/month = 1246 person-days
 3. Rework: $19 * (14.0 + 14.5) = 542$ PDs
- System Integration and Test phase
 1. Integration and test effort = $(71.9/2 + 5.5 + 52.6 + 12.9 + 14.8 + 12.9)$ person-months * 19 working days/month = 2,560 person-days

2. Rework: $(4.6 + 9.2 + 71.9/2)$ person-months * 19 working days/month = 946 person-days

The phase distributions of project effort under different degrees of requirements changes (from 10% to 40%) are calculated following the same procedure. They are summarized in tables 7.7 to 7.11.

Table 7.2. The breakdown of project effort, schedule, and personnel in the Plan and Requirements phase

Activity	Effort (person-month)	Schedule (month)	FSWP
Requirements Analysis	20.8	6.0	3.5
Product Design	8.1	6.0	1.3
Programming	2.5	6.0	0.4
Test Planning	1.8	6.0	0.3
Verification and Validation	3.5	6.0	0.6
Project Office	5.8	6.0	1.0
CM/QA	1.4	6.0	0.2
Manuals	2.3	6.0	0.4

Table 7.3. The breakdown of project effort, schedule, and personnel in the Product Design phase

Activity	Effort (person-months)	Schedule (month)	FSWP
Requirements Analysis	14.0	7.4	1.9
Product Design	45.9	7.4	6.2
Programming	15.1	7.4	2.1
Test Planning	6.7	7.4	0.9
Verification and Validation	8.4	7.4	1.1
Project Office	11.2	7.4	1.5
CM/QA	2.8	7.4	0.4
Manuals	7.8	7.4	1.1

Table 7.4. The breakdown of project effort, schedule, and personnel in the Programming phase

Activity	Effort (person-month)	Schedule (month)	FSWP
Requirements Analysis	14.5	12.0	1.2
Product Design	29.0	12.0	2.4
Programming	204.7	12.0	17.1
Test Planning	19.9	12.0	1.7
Verification and Validation	30.8	12.0	2.6
Project Office	21.7	12.0	1.8
CM/QA	23.6	12.0	2.0
Manuals	18.1	12.0	1.5

Table 7.5. The breakdown of project effort, schedule, and personnel in the Integration and Test phase

Activity	Effort (person-month)	Schedule (month)	FSWP
Requirements Analysis	4.6	7.9	0.6
Product Design	9.2	7.9	1.2
Programming	71.9	7.9	9.1
Test Planning	5.5	7.9	0.7
Verification and Validation	52.6	7.9	6.6
Project Office	12.9	7.9	1.6
CM/QA	14.8	7.9	1.9
Manuals	12.9	7.9	1.6

Table 7.6. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 0%

Activity \ Phase	Requirements	Development	SIT
Specification	583	-	-
Development	201	7,222	-
Integration and Test	-	-	2,543
QA	93	1,246	-
Rework	-	542	945

Table 7.7. CSE-SD-equivalent activity distribution of effort
(person-months) by phase: BRAK = 10%

Activity \ Phase	Requirements	Development	SIT
Specification + Rework	652	-	-
Development	224	8,067	-
Integration and Test	-	-	2,857
QA	103	1,391	-
Rework	-	604	1,055

Table 7.8. CSE-SD-equivalent activity distribution of effort
(person-months) by phase: BRAK = 20%

Activity \ Phase	Requirements	Development	SIT
Specification + Rework	718	8,919	-
Development	249	-	-
Integration and Test	-	-	3,157
QA	114	1,537	-
Rework	-	665	1,169

Table 7.9. CSE-SD-equivalent activity distribution of effort
(person-months) by phase: BRAK = 25%

Activity \ Phase	Requirements	Development	SIT
Specification	754	-	-
Development	260	9,344	-
Integration and Test	-	-	3,312
QA	120	1,609	-
Rework	-	699	1,226

Table 7.10. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 30%

Activity \ Phase	Requirements	Development	SIT
Specification + Rework	789	9,774	-
Development	272	-	-
Integration and Test	-	-	3,467
QA	125	1,685	-
Rework	-	732	1,282

Table 7.11. CSE-SD-equivalent activity distribution of effort (person-months) by phase: BRAK = 40%

Activity \ Phase	Requirements	Development	SIT
Specification + Rework	859	-	-
Development	296	10,648	-
Integration and Test	-	-	3,775
QA	135	1,835	-
Rework	-	798	1397

7.2.3 Calibrate CSE-SD Against COCOMO

We next calibrate CSE-SD to produce similar project behaviors as those of COCOMO for the BASELINE project, including the breakdown of project effort, schedule, and full-time-equivalent software personnel.

To produce similar software personnel distribution patterns, we adjust the values of two CSE-SD parameters: *planned WF* (the originally planned work force) and *staffing plan stability* (the degree that project management stays with the original staffing plan). These two parameters, together with the desired work force level (*target WF*) as determined in the *Project Control* sector, determine the project staff level needed to complete the project on the scheduled completion date.

The *planned WF* parameter, as depicted in figure 7.1, shows the original staffing plan as a function of project development time. For example, at the beginning of the project (Time = 0), the planned full-time-equivalent software personnel is six. The *staffing plan stability* parameter is modeled as a function of the ratio of project time remaining and WF production delay (average time to hire and assimilate new staff members), as illustrated in figure 7.2. For example, if the WF production delay is 120 working days, and there are 600 working days remaining to complete the project (i.e., the project time remaining/WF production delay ratio is $600/120 = 5$), then the value of the *staffing plan stability* parameter is 1. In other words, management will stay with the original staffing plan (i.e., the project staffing plan is stable). However, when the value drops below 1, management will consider changing the original staffing plan and either hire new people or transfer staff members out of the project, depending on the actual progress of the project.

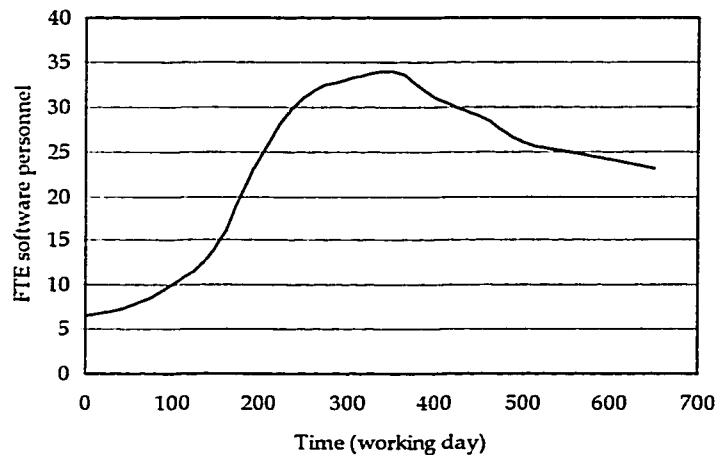


Figure 7.1. Planned work force distribution.

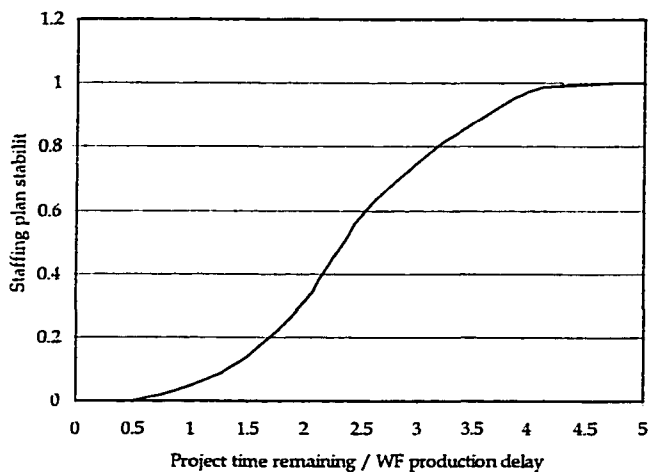


Figure 7.2. Staffing plan stability.

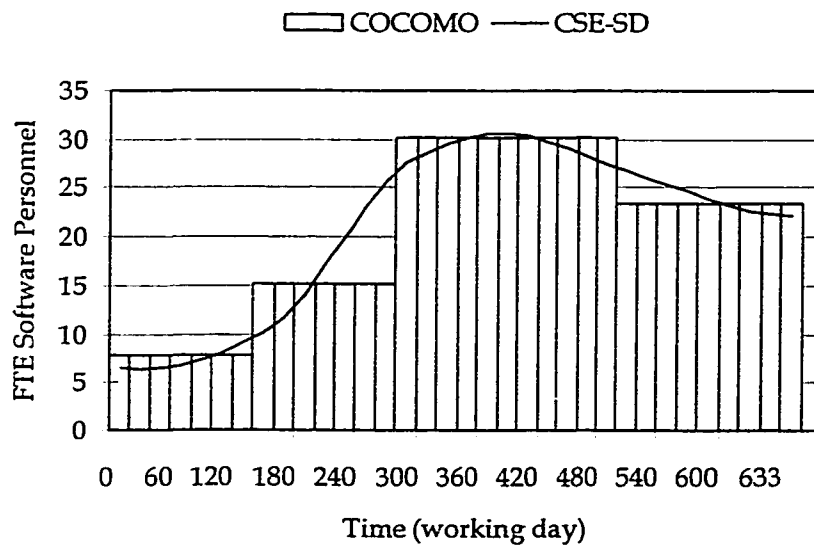


Figure 7.3. Comparison of FTE software personnel distribution.

Another thing to consider is to adjust staff members' average productivity to match COCOMO. In CSE-SD, a requirement unit is assumed to be 125 LOC large; a

development unit is assumed to be 60 LOC large [7]. The total number of requirements and development units is 1,024 (128,000/125) and 2,134 (128,000/60), respectively. The average productivities of different activities are:

- Requirements specification: $1,024/583 = 1.76$ requirements per person-day.
- Development: $2,134/(7,222+201) = 0.287$ development units per person-day.
- Integration and testing: $2,134/2,543 = 0.839$ units integrated and tested per person-day.

Table 7.12 and figure 7.4 show a close resemblance between the data generated from CSE-SD and those of COCOMO for the BASELINE project with 0% requirements changes. For different degrees of requirements changes, we follow the same procedure to produce a similar behavior for the BASELINE project. The biggest percentage difference in project effort between COCOMO and CSE-SD is less than 1.2% (comparing the COCOMO column and the C1xR1xD1 column in table 7.13). By calibrating CSE-SD against COCOMO under different degrees of requirements changes to produce similar nominal project behaviors, we are more confident about the data generated from CSE-SD when we change the values of the two manpower allocation parameters (*fraction daily manpower to Requirements phase* and *frac dev manpower to SIT*) to simulate different degrees of phase overlapping.

Table 7.12. Comparison of project effort (person-days)

	COCOMO	CSE-SD	% difference
Project effort	13,393	13,366	0.22%
- Requirements	676	681	0.74%
- Development	9,211	9,246	0.38%
- Integration and test	3,506	3,440	1.38%
Project duration	633	635	0.16%

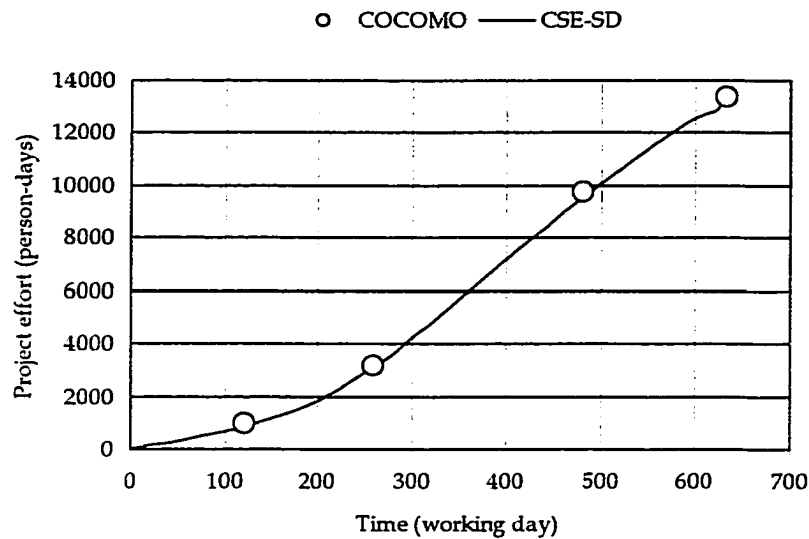


Figure 7.4. Comparison of cumulative project effort.

7.3 Impact of Phase Overlapping

Three counteracting factors determine the outcome of a phase overlapping-based software development project: (1) the degree of phase overlapping; (2) the degree of across-phase communication; and (3) the stability of upstream information and downstream sensitivity to changes to the information. As discussed in section 4.2.1, increasing the degree of phase overlapping reduces project development time, because more work is done simultaneously. However, an increased across-phase communication overhead and rework tasks in downstream phase might erase the benefits gained by doing things in parallel. In this section, we want to determine (1) the degree of phase overlapping that has the shortest project development time and (2) the degree of phase overlapping that has the lowest project cost.

7.3.1 Modeling Phase Overlapping

Phase overlapping occurs when activities of different phases are performed at the same time. Phase overlapping means project manpower resource must be allocated to different phases so that activities in different phases can be performed simultaneously. Across-phase manpower allocation is controlled by two parameters: *frac daily MP to reqs phase* and *frac dev MP to SIT*. The *frac daily MP to reqs phase* parameter determines the fraction of the total daily manpower to be allocated to the Requirements phase. The remainder of the manpower, after allocating to the Requirements phase, is shared by the Development phase and System Integration and Test phase. The distribution of the remaining manpower to these two phases is controlled by the *frac dev MP to SIT* parameter.

The two manpower allocation parameters are modeled as graph functions, as shown in figure 7.5. By adjusting the values of these two parameters, we can simulate different degrees of phase overlapping and investigate their impacts on project cost and development cycle time. To examine the impacts of phase overlapping under different degrees and patterns of requirements changes, we select three different representative phase overlapping modes:

- R1 x D1: It represents a nominal COCOMO project.
- R2 x D2: It represents a modest degree of phase overlapping.
- R3 x D2: It represents a high degree of phase overlapping.

The general shapes of R1, R2, R3, D1, and D2 are shown in figure 7.5.

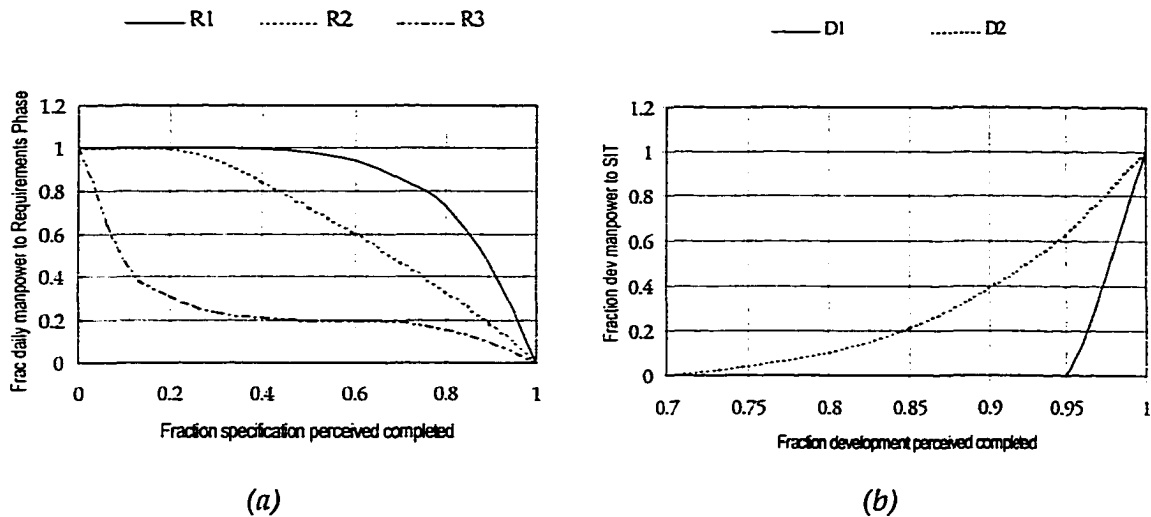


Figure 7.5. Modeling phase overlapping; (a) fraction daily manpower to requirements phase; (b) fraction development manpower to SIT.

7.3.2 Modeling Requirements Changes

The stability of the upstream information (requirements) and the downstream sensitivity to the changes in the exchanged information is another critical factor that determines the outcome of a phase overlapping project. Requirements changes are the major cause of software project delays and cost overruns, especially under the situation of phase overlapping.

When a requirement is changed, you have to alter design to meet the changed requirements. You might have to throw away part of the old design, and, because it has to accommodate existing code, the new design will take longer than it would have without the change. You also have to discard code and test cases affected by the requirement change and write new code and test cases. Even code that is other-

wise unaffected must be retested to make sure the changes in other areas have not introduced any new errors [56].

In CSE-SD, rework overhead is partitioned into two parts: (1) the overhead that results from the increase of development workload; and (2) the overhead incurred by the requirements change to take care of the affected work products. For example, a requirements change is treated as the increase of one unit of regular work plus the overhead to adjust the design, code, test cases, and related documents that are affected by the change, whether it is added, modified, or deleted. With a 30% increase in project size, the project effort is expected to be 30% higher than that without a requirements change (13,393 person-days for the BASELINE project). Without considering the rework overhead, a project 128 KLOC large with a 30% requirements change is expected to need $1.3 * 13,393 = 17,411$ person-days. COCOMO estimate of project effort with 30% requirements changes is 18126 person-days. The difference between 18,126 person-days and 17,411 person-days (i.e., project effort without considering rework overhead) is 715 person-days (this is the rework overhead).

Rework overhead is captured in the *Change Rework Overhead* model parameter. Change rework overhead is accumulated at the rate of *daily MP to change rework*, as determined by three parameters: *nominal rework overhead*, *rework cost ratio*, and *daily MP factor*. For example, if project staff members spend 50% of their daily time on project-related production work, then a requirements change with 0.5 person-day nominal rework overhead will cause them to spend one full day (i.e., $0.5/50%$) to rework all affected work products.

On large projects, the cost to rework a requirements during architecture design is typically five times as expensive to rework as it would be if it were done during the requirements analysis phase; during coding; it is 10 times as expensive; during unit or system test, it is 20 times as expensive ([22], as cited in [56]). The rework cost

ratio is shown in figure 7.6. The rework cost due to a requirements change is determined by multiplying nominal rework overhead by the rework cost ratio.

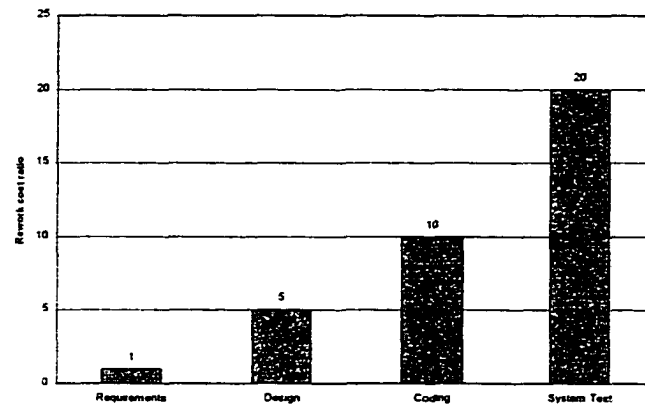


Figure 7.6. Rework cost ratio.

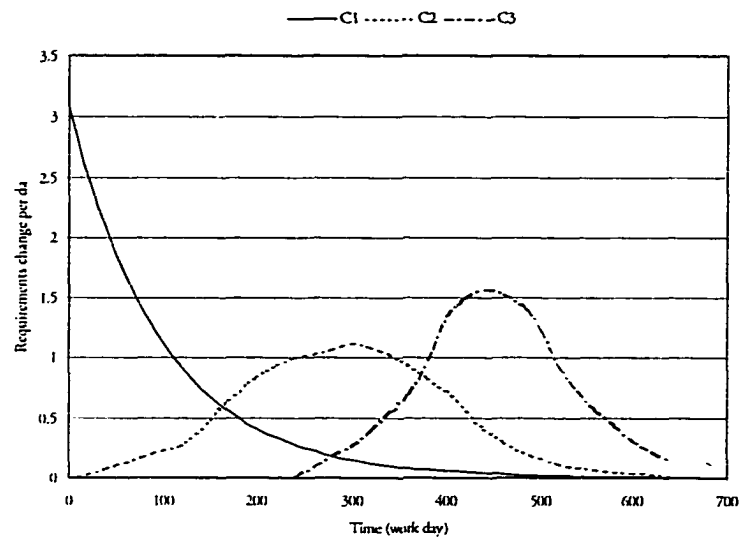


Figure 7.7. Three patterns of requirements change.

Figure 7.7 shows three different requirements change patterns. These three curves represent different patterns of how system requirements are stabilized. Curve C1 represents the situation where most of the requirements changes occur in the requirements analysis phase. Curves C2 and C3 indicate the situations where most of the requirements changes occur in the design and coding phases, respectively. By combining these three requirements change patterns with the three different modes of phase overlapping (i.e., R1xD1, R2xD2, and R3xD2), we can simulate different project scenarios to assess the impact of the phase overlapping concurrent development approach.

7.3.3 Simulation Results

After calibrating CSE-SD against COCOMO, we use the data of the BASELINE project as a reference to examine the effects of the phase overlapping concurrent development approach. We perform nine sets of simulations (from the C1xR1xD1 combination to the C3xR3xD2 combination) for each level of requirements changes, ranging from 10% to 40% requirements changes.

Figure 7.8 illustrates the effects of the Phase Overlapping concurrent development approach on project development cycle time. The results of the three different phase overlapping modes are summarized in tables 7.13 to 7.15. The same results are depicted in figure 7.10. Among the nine combinations of requirements change patterns and phase overlapping that we examine (from C1xR1xD1 to C3xR3xD2), the C1xR3xD2 combination has the shortest project development cycle and lowest project cost. For example, the shortest project development cycle for a 128 KLOC project with 20% requirements changes is 649 (marked with *) working days. The lowest project cost is 15,547 person-days.

Our simulated results show that when most of the requirements changes occur during the requirements analysis phase (the C1xRxD curves), phase overlapping can improve the development process both by reducing project effort and development cycle time. For example, the aggressive (R3xD2) phase overlapping mode helps to cut the project development cycle time from 682 working days to 649 working days, which is about a 4.8% ($33/682$) improvement even when the requirements change is 20%, as long as the requirements changes occur in the requirements analysis phase. Attempting a higher degree of phase overlapping under the same project situation also reduces project effort. The savings in this case is about 5.9% (from 16528 person-days to 15547 person-days)

When most of the requirements changes occur during the product design phase or later, phase overlapping may not be helpful. For example, the modest degree of phase overlapping (R2xD2) reduces project development cycle time only when requirements change is below 30%. Aggressive phase overlapping (R3xD2) is helpful only when requirements change is below 10%. In both cases, the improvements in project development cycle time are not significant. On the other hand, software project managers have to pay the price of increased project effort in attempting phase overlapping.

As predicted, late requirements changes cause project duration and cost to increase, irrespective of the degree of phase overlapping. The percentage increase in project cost and development cycle time ($100 \times (C3 - C1) / C1$) due to late requirements changes under different degrees of requirements changes are shown in figure 7.8 and 7.9, respectively.

When requirements changes exceed 25%, the R2xD2 case (modest degree of phase overlapping) is less sensitive to late requirements changes in terms of project duration increase than the R1xD1 case (nominal case). The R2xD2 case has a 10%

duration increase as opposed to the 15% increase in the R1xD1 case when requirements changes is 40%.

The effort penalty due to late requirements changes shows a slightly different trend. All three phase overlapping cases, including the nominal case, display similar project effort increase patterns, especially when the requirements change is below 10%. Under all situations, the R1xD1 case (nominal case) is least sensitive to late requirements changes in terms project effort increase. Our results show that the R2xD2 case (modest degree of phase overlapping) is least sensitive to late requirements changes in terms of project duration increase and the R1xD1 case is least sensitive to project effort increase.

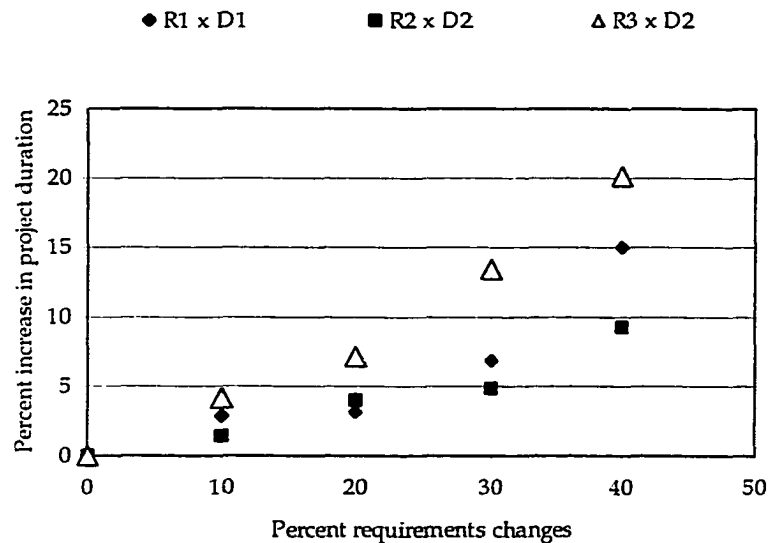


Figure 7.8. Project duration increase due to requirements changes.

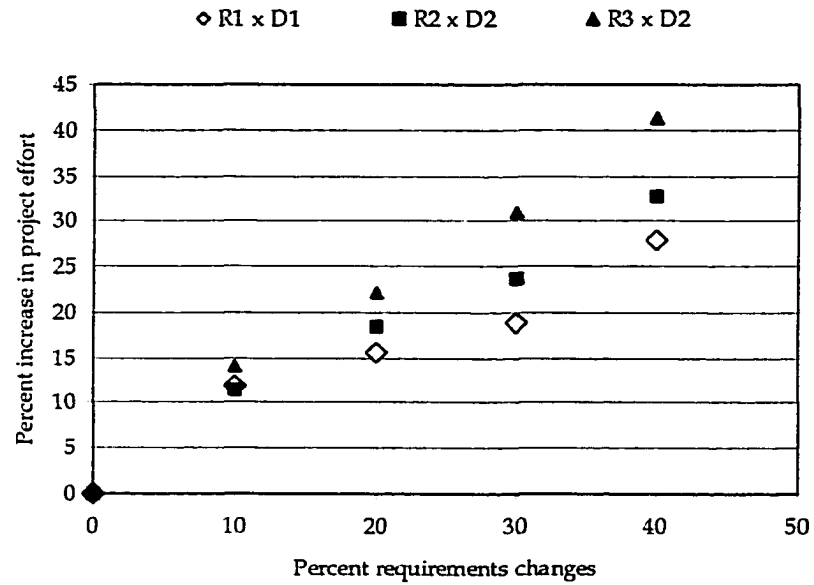


Figure 7.9. Project effort increase due to requirements changes.

Table 7.13. Nominal project (R1xD1) with different requirements change patterns

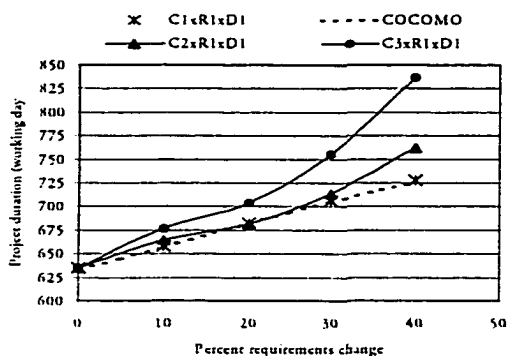
Requirements change	COCOMO	C1xR1xD1	C2xR1xD1	C3xR1xD1
0%	633 (13,393)	635 (13,366)	635 (13,366)	635 (13,366)
10%	657 (14,951)	658 (14,820)	664 (15,757)	677 (16,569)
20%	682 (16,528)	682 (16,366)	681 (17,566)	704 (18,923)
30%	705 (18,126)	706 (17,965)	714 (19,738)	755 (21,341)
40%	726 (19,743)	728 (19,520)	762 (22,681)	837 (24,967)

Table 7.14. Modest phase overlapping (R2xD2) with different requirements change patterns

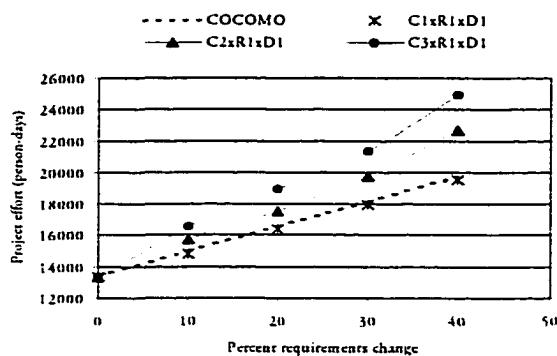
Requirements change	COCOMO	C1xR2xD2	C2xR2xD2	C3xR2xD2
0%	633 (13,393)	618 (12,951)	618 (12,951)	618 (12,951)
10%	657 (14,951)	645 (14,502)	653 (15,495)	654 (16,167)
20%	682 (16,528)	668 (16,027)	674 (17,435)	695 (19,207)
30%	705 (18,126)	697 (17,707)	702 (19,445)	731 (21,916)
40%	726 (19,743)	720 (19,273)	747 (22,312)	787 (25,581)

Table 7.15. Aggressive phase overlapping (R3xD2) with different requirements change patterns

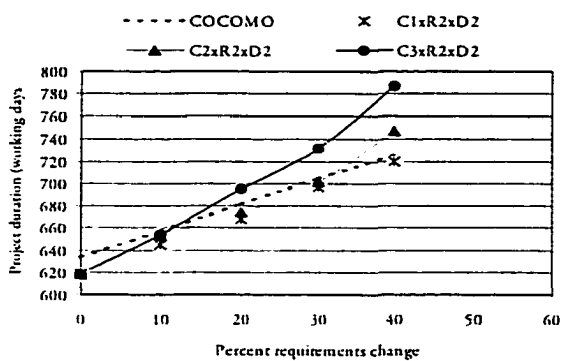
Requirements change	COCOMO	C1xR3xD2	C2xR3xD2	C3xR3xD2
0%	633 (13,393)	614 (12,879)	614 (12,879)	614 (12,879)
10%	657 (14,951)	628 (14,078)	642 (14,934)	654 (16,050)
20%	682 (16,528)	649 (15,547) *	678 (16,979)	695 (19,006)
30%	705 (18,126)	679 (17,231)	710 (19,090)	770 (22,558)
40%	726 (19,743)	707 (18,893)	774 (22,699)	850 (26,699)



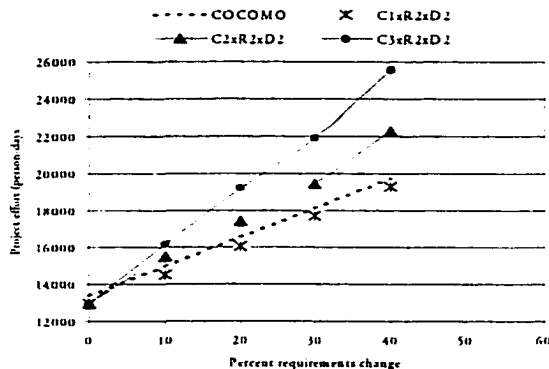
(a)



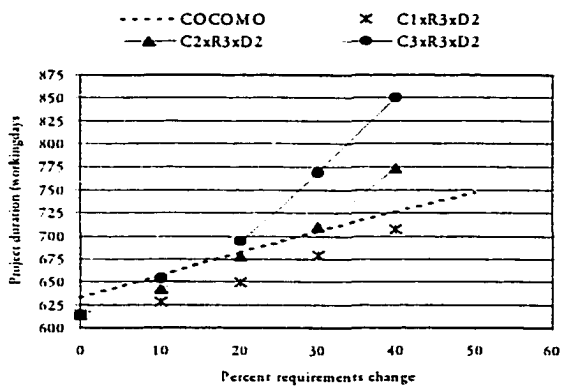
(b)



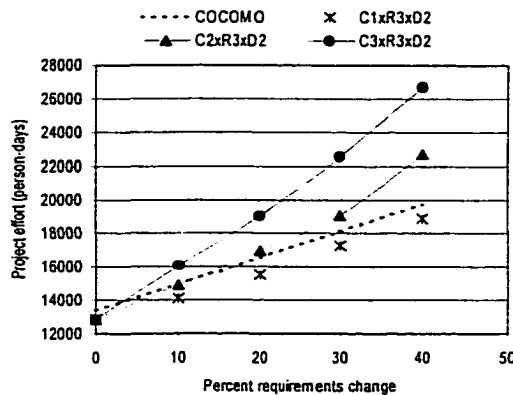
(c)



(d)



(e)



(f)

Figure 7.10. The effects of phase overlapping on project effort and development cycle time; (a) Project duration (R1xD1); (b) Project effort (R1xD1); (c) Project duration (R2xD2); (d) Project effort (R2xD2); (e) Project duration (R3xD2); (f) Project effort (R3xD2).

7.4 Impact of Synchronous Concurrent Subsystems

In this section, we assess the impact of the SCS concurrency with a focus on two questions: (1) Is the SCS concurrency a feasible approach? Will it reduce project effort and development cycle time? and (2) What is the optimal number of subsystems (subteams) that leads to the lowest project effort and shortest development cycle time?

As discussed in section 4.2.2, three counteracting factors are critical in determining the outcome of a SCS project, namely, how the project is decomposed (i.e., the number of subsystems), the incurred communication overhead due to project decomposition, and the incurred extra rework due to interteam problems.

Grouping developers into teams affects the overall communication overhead. Consider the case of grouping N developers into t equal-sized teams of n (i.e., N/t) members per team. The possible number of communication links is the sum of the number of interteam communication links plus the number of intrateam communication links. The possible number of communication links among t teams is $t(t-1)/2$, and the number of communication links among n members within a team is $n(n-1)/2$. Since there are t teams, the total number of intrateam communication links is $(tn)(n-1)/2$. The interteam and intrateam communication overheads increase in proportion to t^2 and tn^2 , respectively.

Breaking a single large team into multiple smaller teams decreases the amount of intrateam communication overhead. However, for a given number of developers, increasing the number of concurrent teams will increase the amount of interteam communication overhead. It is critical to determine an optimal number of concurrent subteams to minimize the overall communication overhead.

7.4.1 Determining Communication Overhead

The overall average time that project staff members spend on communicating with other members of the project each day is captured in the *overall communication overhead* parameter (shown in middle-left of figure 7.11). We classify communication overhead into two categories: communications within teams (*intrateam communication overhead*) and communications across teams (*interteam communication overhead*). Communications within a team usually are frequent and informal. Communications across teams usually are more formal and via meetings and/or documented agreements. A well-partitioned project usually has a higher level of communication traffic within a team than across teams.

Both the *intrateam communication overhead* and the *interteam communication overhead* parameters are modeled as a graph function, as shown in figure 7.12. The *intrateam communication overhead* is a function of average team size, while the *interteam communication overhead* is modeled as a function of the number of teams. The general shapes of the two graph functions are based on the assumption that communication overhead depends on the number of communication links ([7], [22]).

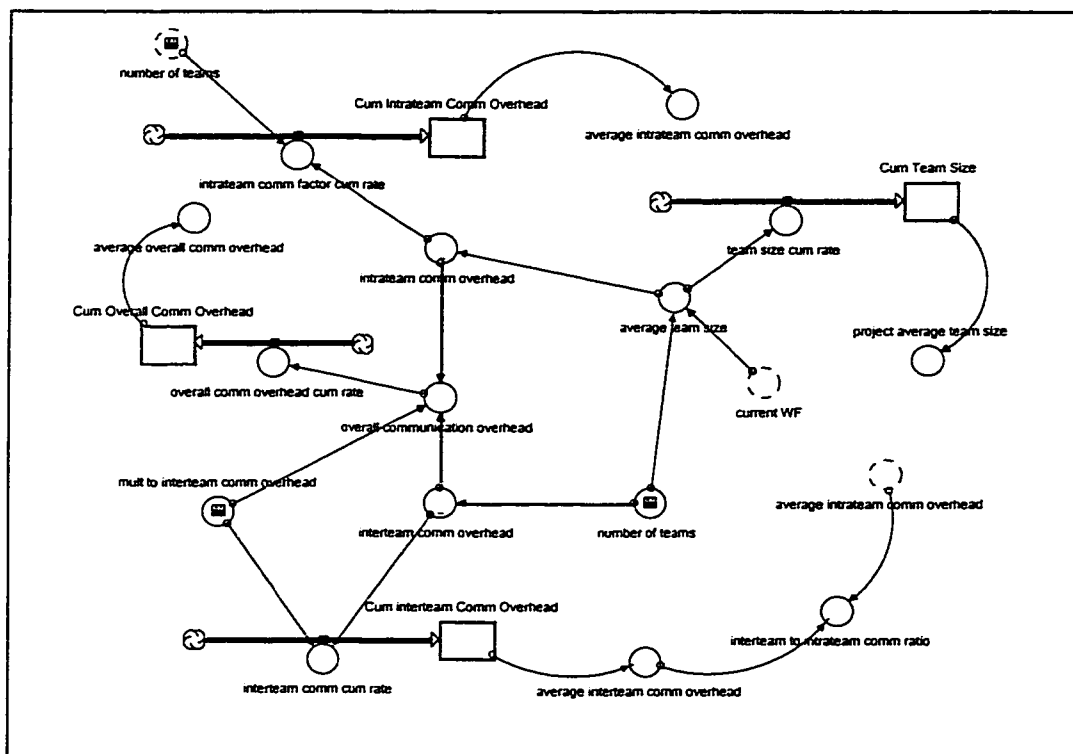


Figure 7.11. Determining the overall communication overhead.

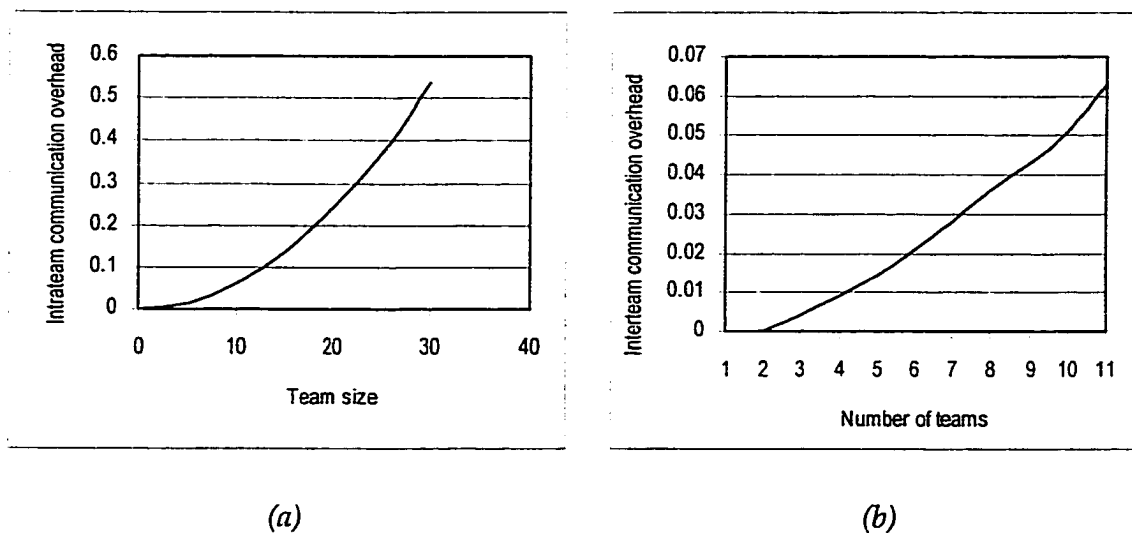


Figure 7.12. Intrateam and interteam communication overheads; (a) Intrateam communication overhead; (b) Interteam communication overhead.

7.4.2 Interteam Interactions

Breaking a large team into subteams reduces the communications flow, but the risk of problems caused by isolated concurrent works grows. Some aspect of one team's work may impact work being done by another [42]. Teams involved in concurrent development of different subsystems (e.g., hardware components and software components) must have a steady flow of information among the groups to prevent potential integration problems [21]. As Aoyama notes [19]:

Multiple teams working on the related enhancements may disrupt the system's integrity. In requirements specifications, for example, this can cause inconsistent and/or incomplete specifications. In design and implementation, simultaneous updates to a single module may violate the modules' consistency.

We define an "interference" as an interteam problem that is caused by multiple concurrent development teams and could have been avoided if the project was done

by one team. In requirements analysis phase, for example, interferences could mean conflicting requirements, missing requirements, or duplicate requirements. Although these problems also exist in a one-team Waterfall process, they have different meanings here. When the workload is assigned to different teams, missing requirements mean that no team takes charge of those requirements; duplicate requirements means that at least two teams work on the same requirements; and conflicting requirements mean that different teams have different interpretations of the same requirements.

Interferences among requirements specifications are at a higher level than those encountered during design and implementation. In requirements analysis phase, interferences are intangible and created as specifications are elaborated. Without ongoing, informal communication, simultaneous work on different components of a project will create chaos rather than progress and will consume more time than the sequential approach [65].

Interteam interferences amplify along two dimensions: the “degree of concurrency” dimension and the “development life cycle” dimension. Obviously, if there is only one team, there would be no interteam interferences. However, as the number of development teams increases, interteam interferences will grow, and worse yet, in non-linear manner. The relationship between the number of interteam interferences generated and the number of teams is modeled as the *across-team interference amplification* parameter; its general form, as depicted in figure 7.13 (a), is based on our discussions with Mikio Aoyama and three other Fujitsu project managers [20].

Interteam interferences also grow along the development life cycle dimension. An upstream interference amplifies more downstream interferences when downstream activities work on the upstream interference. The newly generated design interferences, in turn, will generate more coding interferences. The longer the

interference remains undetected, the more downstream interferences will be amplified. For example, an inconsistent requirements specification (i.e., a specification interference) will amplify one or more design interferences. If a requirement is associated with five design units, then one requirements interference will amplify five design interferences.

Interference amplification within the development phase (including design and coding) is modeled as the *dev phase interference amplification* parameter (defined in the *Interteam Interactions* sector). The general shape of the *dev phase interference amplification* parameter, as depicted in figure 7.13 (b), is based on the experience of Fujitsu [20]. In the initial stage of the development phase, a design interference will, on average, amplify two-and-a-half downstream interferences (i.e., detailed design and coding interferences). As the development phase progresses to the end, all interferences are coding interferences, and therefore will not amplify more interferences (the value of the *dev phase interference amplification* parameter approaches 0).

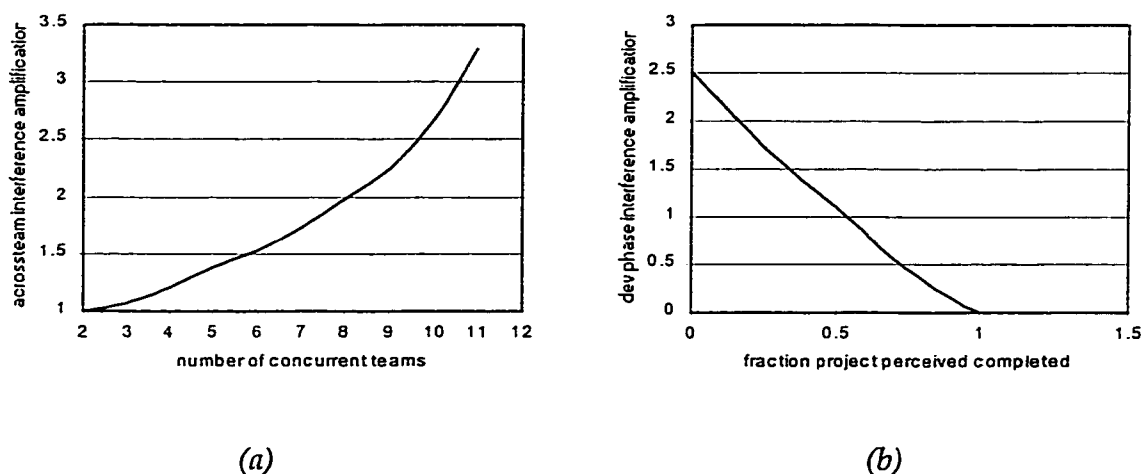


Figure 7.13. Interteam interference amplification; (a) Across team interference amplification; (b) Development phase interference amplification.

In our model, we assume that interteam interferences are detected via interteam QA and system integration activities. The impact of interteam communication on interference detection is implicitly included in four parameters: *frac reqs int* (number of interferences committed per requirements specification), *frac dev int* (number of interferences committed per unit developed), *dev phase interference amplification* (number of coding interferences amplified per design interference), and *across team interference amplification* (multiplier to interference amplification due to an increase of concurrent teams). Effective interteam communication will have smaller values for the four parameters. The effort spent in interteam QA activities is modeled as the *daily MP on int detection* parameter (i.e., the amount of daily manpower allocated to interference detection).

Detection of interteam interferences results in respecifying, redesigning, recoding, and retesting. In the Fujitsu's concurrent development project [14], interteam technical reviews (specification/design review and code inspection) are conducted at the end of each life cycle phase. Interteam technical reviews are one-day workshops that involve team leaders reviewing completed work to locate interteam interferences.

7.4.3 Experimentation Setting

We select three representative patterns of interteam-to-intrateam communication ratio to cover different situations, from the "light interteam communication" situation (M1) and the "medium interteam communication" situation (M2) to the "high interteam communication" situation (M3). For example, as shown in figure 7.14, if a project is partitioned into eight subsystems concurrently being developed by eight subteams, M1 represents the situation in which the average interteam

communication overhead is about 25% ($CR = 0.25$) of the intrateam communication overhead; M2 represents the situation in which the project has a balanced interteam and intrateam communication overhead (i.e., $CR = 1$); M3 represents the situation in which the across-team communication traffic is about twice heavier than the intrateam communication traffic (i.e., $CR = 2$).

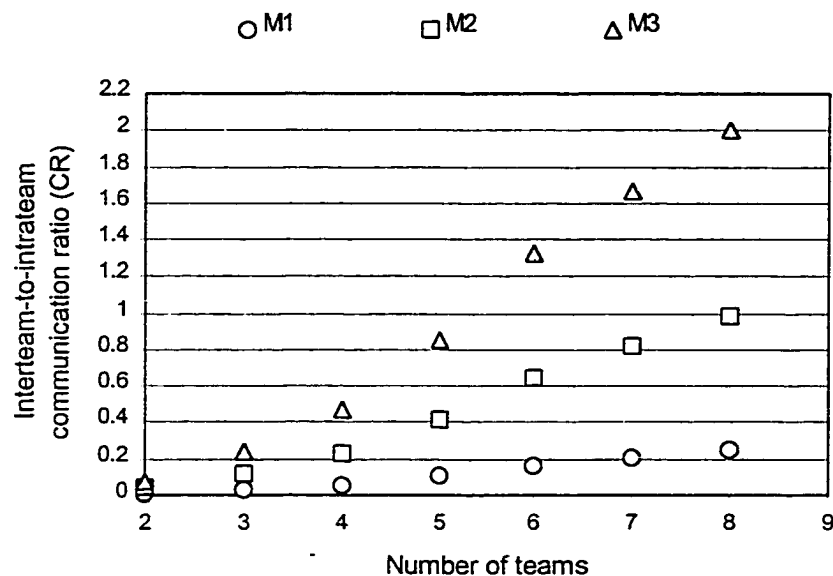


Figure 7.14. Interteam-to-intrateam communication ratio.

The resolution of interteam problems (i.e., interteam interferences) results in respecifying, redesigning, recoding, and retesting the work that has been done. The amount of extra rework incurred by concurrent development definitely has an impact on project cost and development cycle time. As with the communication ratio, we select three representative patterns-F1, F2, and F3-to model different degrees of extra rework (represented as a percentage of the original planned work),

from modest degree (F1) and medium degree (F2) of rework to high degree of rework (F3). For example, as shown in figure 7.15, if a project is divided into eight subsystems concurrently being developed by eight subteams, F1 represents the situation of 25% rework; F2 represents the situation of 50% rework; and F3 represents the situation in which rework incurred by concurrent development is 75%.

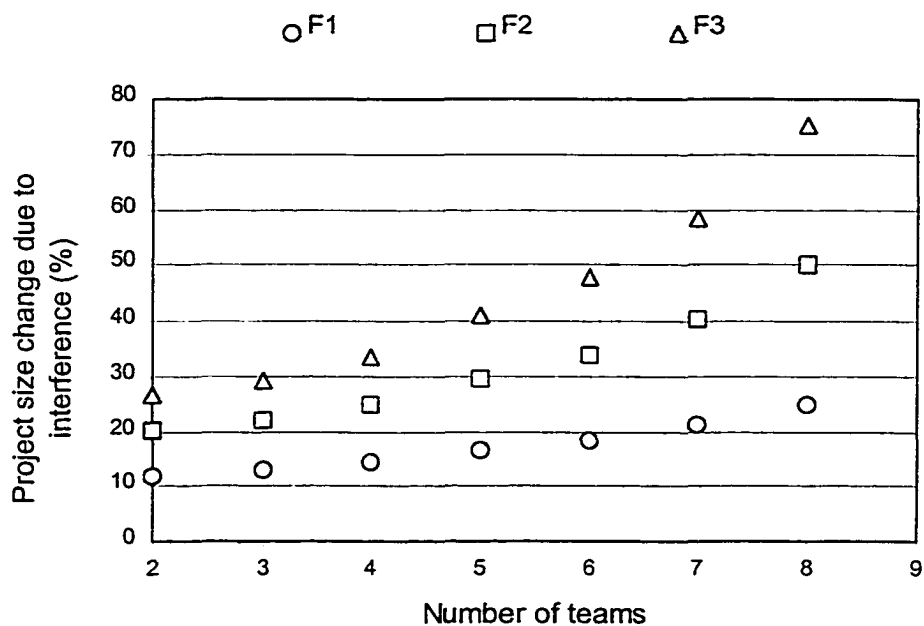


Figure 7.15. Project size change due to resolution of interteam interferences.

To perform a systematic and comprehensive assessment of the SCS (synchronous concurrent subsystems) development approach under different project scenarios, we conduct nine sets of simulation runs using the nine "FxM" combinations for each number of concurrent teams, from one to eight. The meanings of the nine FxM combinations are explained as follows:

- (1) M1xF1: Low communication ratio combined with modest degree of rework caused by interteam interferences. This is the best-case scenario for an SCS project. Possible reasons that an SCS project exhibits this type of behavior are the project is well-partitioned or subsystems are loosely related. Under these conditions, the need for interteam communication is minimal.
- (2) M1xF2: Low communication ratio combined with medium degree of rework due to interteam interferences. Projects may not be perfectly partitioned, and teams do not communicate enough to resolve and prevent interteam problems.
- (3) M1xF3: Low communication ratio combined with high degree of rework due to interteam interferences. Projects are not well-partitioned, subsystems are tightly-coupled. Teams do not communicate enough to coordinate their work. Therefore, the incurred rework is high.
- (4) M2xF1: Medium communication ratio combined with modest degree of rework due to interteam interferences. Projects may not be perfectly partitioned, however, teams maintain a certain level of communication to coordinate their work and prevent future interteam problems from occurring. Therefore, the incurred rework is minimal.
- (5) M2xF2: Medium communication ratio combined with medium degree of rework due to interteam interferences. Projects may not be perfectly partitioned. Teams do communicate to coordinate their work. However, a certain level of rework to resolve interteam problems is still needed.
- (6) M2xF3: Medium communication ratio combined with high degree of rework due to interteam interferences. Projects are not well-partitioned, and subsystems are tightly coupled. Teams do communicate to coordinate their work. However, the communication might not be effective. Therefore, the incurred rework is still high.

- (7) M3xF1: High communication ratio combined with modest degree of rework due to interteam interferences. Projects may not be perfectly partitioned, however, teams maintain a high level of communication to coordinate their work and prevent future interteam problems from occurring. Therefore, the incurred rework is minimal. Microsoft's Daily Build practice is an example of this type of SCS development.
- (8) M3xF2: High communication ratio combined with medium degree of rework due to interteam interferences. Projects may not be perfectly partitioned. Teams do frequently communicate to coordinate their work. However, the communication may not be effective, and a certain level of rework to resolve interteam problems is still needed.
- (9) M3xF3: High communication ratio combined with high degree of rework due to interteam interferences. This is the worst-case scenario for the SCS development approach. The project is not well-partitioned, and subsystems are tightly coupled, requiring intensive communication and information traffic across subsystem teams.

7.4.4 Simulation Results

Figures 7.16 and 7.17 depict the simulation data of the BASELINE project under twenty-four different project settings. All twenty-four simulation runs simulate projects with M1 (i.e., low interteam-to-intrateam communication ratio) behavior. Three immediate observations can be derived from the two figures.

First, for a given project setting, there exists an optimal number of concurrent teams that leads to lowest project effort and shortest development cycle time. Our results show that, for a 128 KLOC project (without requirements change) with average full-time-equivalent software personnel of 24.2 (data derived from COCOMO

2.0), the optimal number of concurrent teams is four for the M1xF1 combination and three for the M1xF2 and the M1xF3 combinations. The optimal team size for the M1xF1, M1xF2, and M1xF3 combination is six ($24.2/4$), eight ($24.2/3$), and eight ($24.2/3$), respectively.

Second, it is beneficial to organize a project work force into smaller groups. The savings in project effort and development cycle time is most significant from one-team setting to two-team settings. For example, the savings in project effort and development cycle time from one-team setting to the two-team M1xF1 setting is 16.4% (i.e., $(13,329-11,143)/13,329$) and 13.4% (i.e., $(634-549)/634$), respectively. However, the difference between the two-team M1xF1 setting and the four-team M1xF1 (i.e., optimal) setting is not significant. The difference for project effort and development cycle time is only 3.1% (i.e., $(11,143-10,793)/11,143$) and 2.4% (i.e., $(549-536)/549$), respectively.

The second observation can be theoretically justified. For a team with 24 members, the number of potential communication links among team members is 276 (i.e., $24 \times 23 / 2$). The number of potential communication links for two equal-sized teams is 132 (i.e., $2 \times 12 \times 11 / 2$) plus one interteam communication link. The savings is 143 (i.e., $276-133$). When the 24 staff members are grouped into three teams, the number of potential communication link drops to 84 plus three interteam communication links. Now the savings in communication links is only 46 ($133-87$), which is 32% (i.e., $46/143$) of the two-team setting.

Third, it is beneficial to adopt the SCS development approach as long as the incurred extra rework is below a certain threshold value. For example, as depicted in figures 7.16 and 7.17, twenty-two out of twenty-four project settings have benefited from the SCS development approach. The two exceptions are the seven-team M1xF3

and eight-team M1xF3 settings. These two project settings have a 58.5% and 75.3% rework.

The threshold value for the incurred extra rework under the eight-team M1xF3 setting, suggested by CSE-SD, is around 57%. In other words, for an eight-team M1xF3 SCS project setting to be beneficial, the incurred extra rework due to inter-team interferences should not exceed 57%.

The results for the “medium interteam communication” (M2) situation and the “high interteam communication” (M3) situation are depicted in figures 7.18 to 7.21. Like the three M1 settings, there exists an optimal number of concurrent subteams that leads to the lowest project effort and the shortest development cycle time for the M2 and M3 settings. The simulation results show that, for the BASELINE project (without requirements change) with average full-time-equivalent software personnel of 24.2, the optimal number of concurrent subteams is three for the M2xF1, the M2xF2, and the M2xF3 settings. The optimal team size for these three settings is eight ($24.2/3$). The M3 situation exhibits similar behavior. The optimal number of concurrent subteams is three for the M3xF1, M3xF2, and M2xF3 combinations. The optimal team size for all the three M3 combinations is eight ($24.2/3$).

The savings in project development cycle time for organizing project staff into optimal project setting is 15.0% (i.e., $(634-539)/634$) for the M2 situation and 14.5% (i.e., $(634-542)/634$) for the M3 situation. The savings in project effort is more significant than those of development cycle time. The savings is 18.4% (i.e., $(13,329-10,876)/13,329$) for the M2 situation and 17.7% (i.e., $(13,329-10,966)/13,329$) for the M3 situation.

Third, it is beneficial to adopt the SCS concurrent development approach as long as the incurred extra rework is below a certain threshold value. In the M2 (medium interteam communication) situation, it is unwise to attempt a seven-team

concurrent development if the incurred rework is above 57%. As shown in figures 7.18 and 7.19, both the development cycle time (637 working days) and project effort (13,889 person-days) of the seven-team setting is similar to that of the one-team setting (634 working days and 13,329 person-days, respectively). In the M3 (high interteam-to-intrateam communication ratio) situation, our results suggest not organizing project staff into more than six subteams if the incurred rework is above 47%.

In summary, the SCS concurrent development approach is feasible and beneficial. It helps cut project effort and development cycle time. Under sound project conditions (low interteam-to-intrateam communication ratio and low incurred extra rework, i.e., the M1xF1 setting), the SCS development approach cuts project effort by 19% and development cycle time by 15.5%. However, there are limits to the benefits of the SCS development approach. The benefits of the SCS development approach are confined by the relative magnitude of the interteam-to-intrateam communication ratio and the degree of extra rework incurred due to interteam problems.

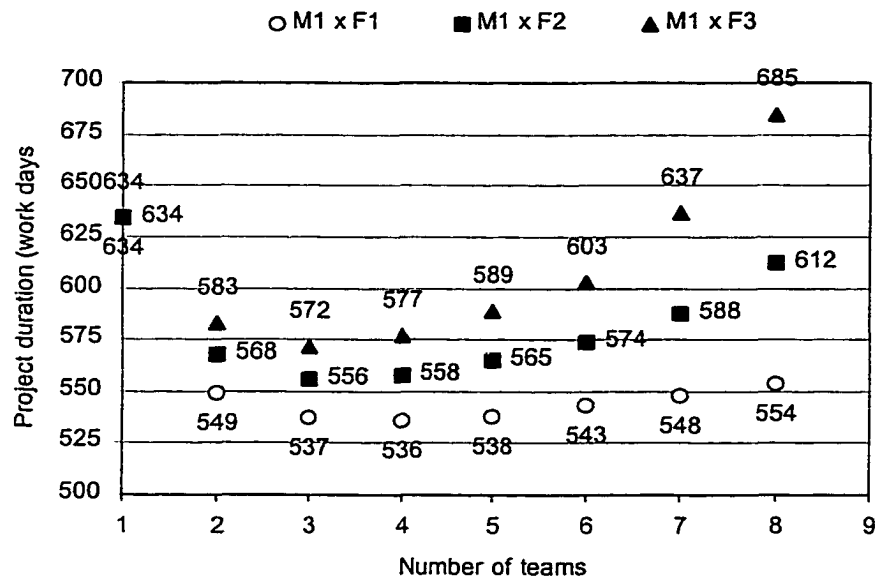


Figure 7.16. Project duration vs. number of teams (low communication ratio M1).

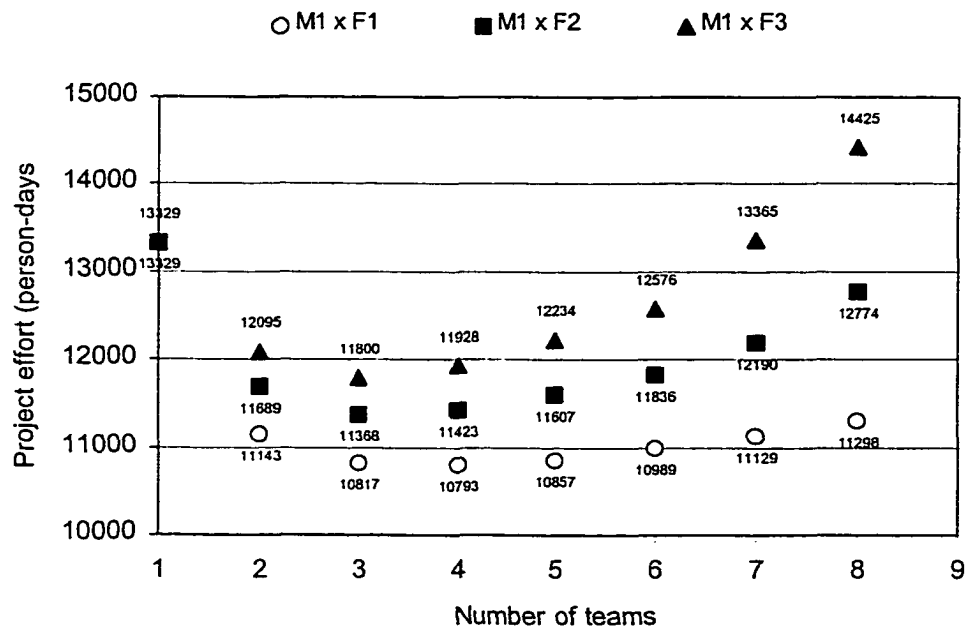


Figure 7.17. Project effort vs. number of teams (low communication ratio M1).

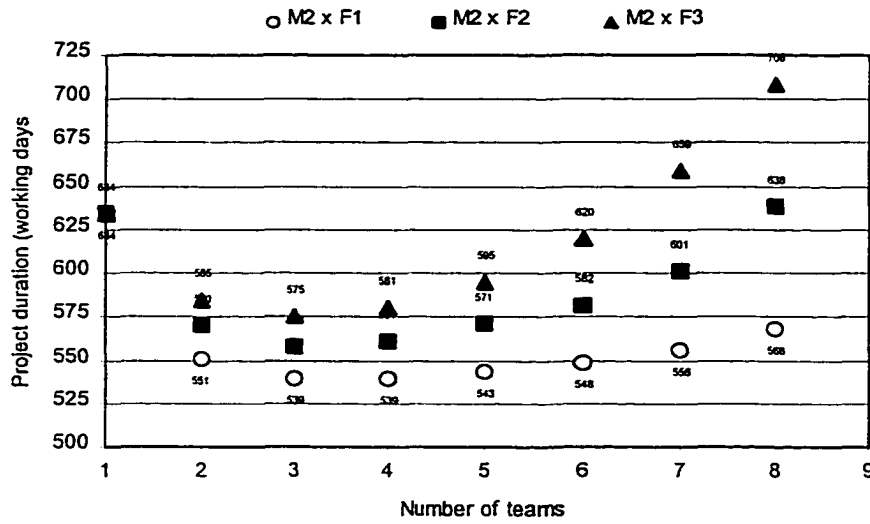


Figure 7.18. Project duration vs. number of teams (medium communication ratio M2).

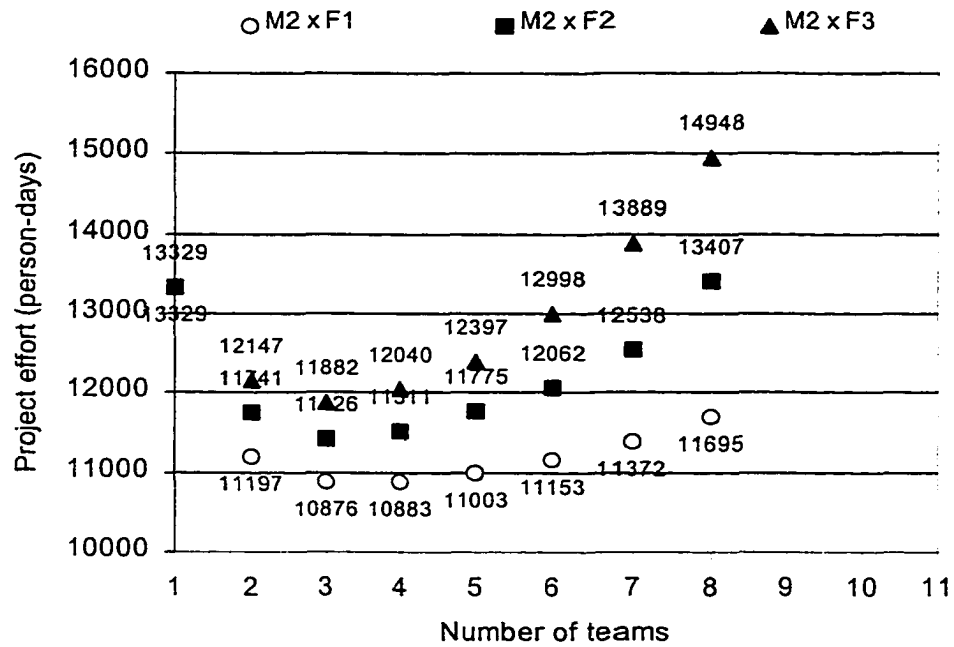


Figure 7.19. Project effort vs. number of teams (medium communication ratio M2).

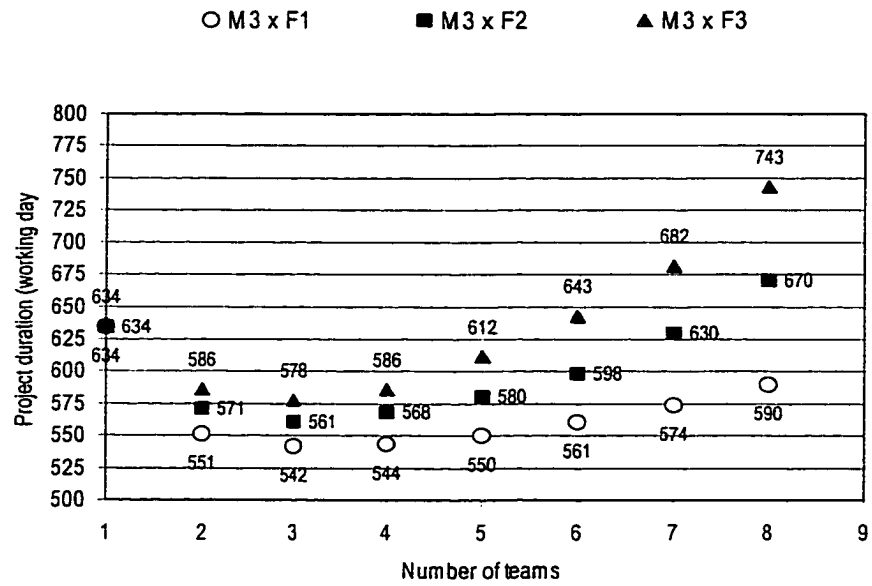


Figure 7.20. Project duration vs. number of teams (high communication ratio M3).

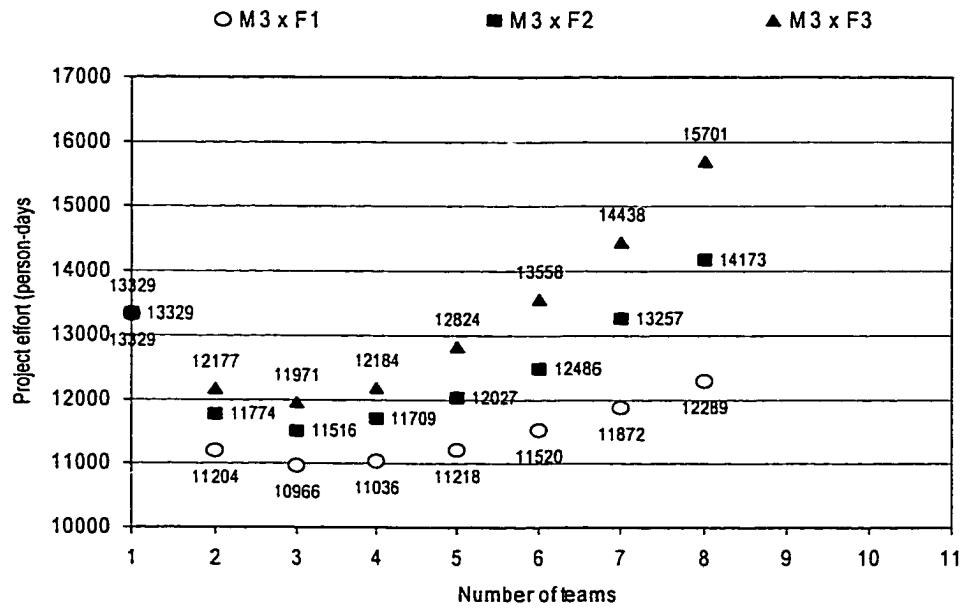


Figure 7.21. Project effort vs. number of teams (high communication ratio M3).

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Contributions of the Research

This research has made three major contributions. First, we have presented a classification of different types of concurrent software engineering (CSE) practices, based on a proposed conceptual Resource-Activity-Work product (RAW) model. The RAW model is able to capture different types of concurrency in different levels of detail. We also have surveyed state-of-the-practice CSE practices and presented them using the RAW model. The RAW representation allows one to easily recognize different types of concurrency that exist in a complex software development process, and, therefore, predict the benefits and potential risks of the development process.

Second, we have identified the specific benefits, potential risks, and the dynamic cause-effect implications of different types of CSE practices. Based on the cause-effect analysis, we have developed a system dynamics simulation model CSE-SD to assess the impact of concurrent software engineering on project cost and development cycle time.

CSE-SD is an economic and effective management policy exploration tool for pre-assessing the benefits and potential risks of reengineering software development processes. It is useful for process definition, process analysis and process redesign. The output of the CSE-SD model provides a predictive reference behavior for the newly reengineered process. The proposed CSE-SD simulation model easily can be extended to assess the impact of other factors.

Third, we have studied three sets of questions using the CSE-SD model: (1) the impact of project restaffing on project cost and development cycle time; (2) the impact of the *phase overlapping* concurrent development approach on project cost and development cycle time; and (3) the impact of the *synchronous concurrent subsystems* development approach on project cost and development cycle time. The results of our study provide strategic information for software project managers who attempt concurrent software product development. The results of our study are summarized in section 8.2.

The utility of the CSE-SD model for a particular organization depends on calibrating it according to local data. While model parameters in CSE-SD are set with reasonable numbers to investigate the impact of CSE practice in general, the results using the defaults will not necessarily reflect all environments.

8.2 Important Findings

Three specific sets of questions have been studied in this thesis: (1) What is the impact of adding people late in a software project? Will the project be completed earlier or be delayed even further as predicted by Brooks' Law? When is the best time to add people to a software project, and how many people should be added? (2) What is the impact of the *phase overlapping* concurrent development approach on project cost and development cycle time? Will phase overlapping reduce project duration and/or cost? What is the optimal degree of phase overlapping in terms of project cost and development cycle time? and (3) What is the impact of the *synchronous concurrent subsystems* (SCS) development approach on project cost and development cycle time? Will the SCS development approach reduce project cost and development cycle time? For a given project, what is the optimal number of subsystems

(subteams) that can lead to the shortest development cycle time and lowest cost? What is the impact of interteam technical review on project duration and cost?

8.2.1 Brooks' Law

We performed an in-depth study of Brooks' Law using the CSE-SD model. The results of the study are based on three sets of simulation runs with different assumptions. First, we use the same assumptions as those of Abdel-Hamid and Madnick (AHM) [7]: (1) project tasks can be partitioned, but there is no sequential constraint among them; and (2) management continuously will add new people as long as it senses a shortage in manpower. Under these assumptions, our results are consistent with those of AHM, namely, adding more people to a late project always causes it to become more costly but does not always cause it to be completed later.

Next, we use a more realistic assumption by considering sequential constraint. We found out that continuously adding people to a late project makes it later and more costly. This confirms Brooks' Law. However, these results are different from those of AHM's. Their results indicated that adding people late in the project (until two calendar weeks remaining to complete the project) will not delay the project. Our results show that, when sequential constraint is significant, adopting such an aggressive manpower acquisition policy causes the project to be delayed further.

Finally, we add another realistic assumption that people are added to a project only once throughout the entire project life cycle, because it is difficult to obtain frequent manpower addition approvals from upper management. We found out that there is an optimal time range for adding people without delaying a project. It ranges from one-third to halfway into the project development. If software project managers cannot make a timely and accurate decision on project restaffing prior to halfway into the project, the project has a high probability of being delayed,

especially when task sequential constraints are involved. However, adding people during the project always causes the project cost to increase.

8.2.2 Impact of Phase Overlapping

Our results show that when 90% of the requirements changes occur during the requirements analysis phase, the *phase overlapping* concurrent development approach reduces both project effort and development cycle time. In other words, if the requirements phase is done well and the requirements specification is fairly complete and stable, then CSE is very helpful. However, when most of the requirements changes occur during the “product design” phase or later, the improvement by CSE in reducing cycle time is not significant. Furthermore, software project managers have to pay the price of increased project effort when attempting the phase overlapping development approach.

Among the nine combinations of “requirements change patterns” and “phase overlapping modes” we examined, the “C1xR3xD2” combination has the shortest project development cycle time and lowest project cost. The “C1xR3xD2” combination represents the situation of attempting aggressive phase overlapping when most of the requirements changes occur during the “requirements analysis” phase.

8.2.3 Impact of Synchronous Concurrent Subsystems

Three important findings are observed from our simulation data. First, for a given project setting, there exists an optimal number of concurrent teams that leads to lowest project effort and shortest development cycle time. For the specific project we studied (i.e., 128 KLOC COCOMO 2.0 nominal project), the optimal number of teams is three if the project is well-partitioned and the amount of rework due to interteam problems is around 30% to 40%. The optimal team size is eight, which is

consistent with that suggested by Graicunias [45]. According to Graicunias, the upper limit of effective staff size is about eight [75].

Second, it is beneficial to organize a project work force into smaller groups. The savings in project effort and development cycle time is most significant from a one-team setting to a two-team setting. For example, the savings in project effort and development cycle time from a one-team setting to the two-team M1xF1 setting (i.e., the combination of low interteam-to-intrateam communication ratio and low rework percentage) is 16.4% and 13.4%, respectively. However, the difference between the two-team M1xF1 setting and the four-team M1xF1 (i.e., optimal number of teams) setting is not significant. The difference for project effort and development cycle time is only 3.1% and 2.4%, respectively.

Third, it is beneficial to adopt the SCS concurrent development approach as long as the incurred extra rework is below a certain threshold value. For example, in the M2 (medium interteam-to-intrateam communication ratio) situation, it is unwise to attempt seven-team concurrent development if the incurred rework is above 57%. In the M3 (high interteam-to-intrateam communication ratio) situation, our results suggest not organizing project staff into more than six concurrent teams if the incurred rework is above 47%.

8.3 Future Work

The proposed CSE-SD model is designed to study the impact of CSE on project cost and development cycle time. It is a comprehensive model that covers the entire software development process, from requirements analysis to system integration and test. However, the proposed model still can be extended to assess the impact of other factors of interest.

As discussed in section 4.2.3, in the asynchronous concurrent subsystems (ACS) concurrency, each subteam evolves its design at a different speed, but their work must be integrated at the end of the project. Therefore, knowing how to control the development progress of each subteam, to be sure they complete their share of work on time, becomes an important issue. Timebox-based project management helps prevent delay of the project by ensuring that no subsystem is late [54]. The proposed CSE-SD model can be extended to assess the impact of instituting timebox management practice when concurrent developments are out of sync.

In section 4.2.4, we identified critical factors in the CFI concurrency, namely, cross-functional integration, empowerment of decision-making authority, co-location of team members, dedicated team members, and setting time as a goal. CSE-SD can be extended to incorporate these factors and test the following hypotheses:

1. Increasing the number of functions represented on the development team decreases development time. Cycle time benefits, however, may diminish, if a cross-function team becomes too large.
2. (a) Decreasing the number of decisions for which approval is required outside the project team decreases development time; (b) Increasing the level of senior management support for the team decreases development time.
3. Setting and measuring fast cycle time as an explicit project goal decreases development time.
4. Co-locating team members decreases development time.
5. As the number of projects to which team members are assigned decreases, development time decreases.

APPENDIX A
CSE-SD MODEL SPECIFICATION

A.1 The Human Resource Subsystem

The *Human Resource* subsystem consists of three sectors: *Work Force*, *Staff Productive Time*, and *Staff Productivity*. They are responsible for modeling the project work force, the amount of time that project staff members actually spend on the project, and their production rate, respectively.

A.1.1 The Work Force Sector

The *Work Force* sector, as shown in figure A.1, keeps track of the current number of project staff members that are working on the project (*current WF*). We divide the available work force into two categories, new staff members (*New Staff*) and experienced staff members (*Exp Staff*), mainly for three reasons. First, new staff members usually are less productive because of their lack of project experience and knowledge. Second, new staff members usually spend part of their time in training and orientation right after they are brought into the project. Training also consumes part of the experienced staff members' productive time. The third reason is that new staff members are prone to commit more errors than the experienced staff members.

Management decides on the number of engineers to hire (*desired new staff*) and/or the number of staff members to bring from other projects (*Desired In Trans Staff*). The hiring and transferring of project staff members take time. The time that it takes to hire new staff members and transfer staff members into and out of the project from within the organization, is modeled as *hiring delay*, *in trans delay*, and *out trans delay*, respectively.

Once the desired number of new work force members is brought into the project, they usually will go through a training/assimilation period before they become experienced and productive. The training/assimilation period is modeled as *assimilation delay*.

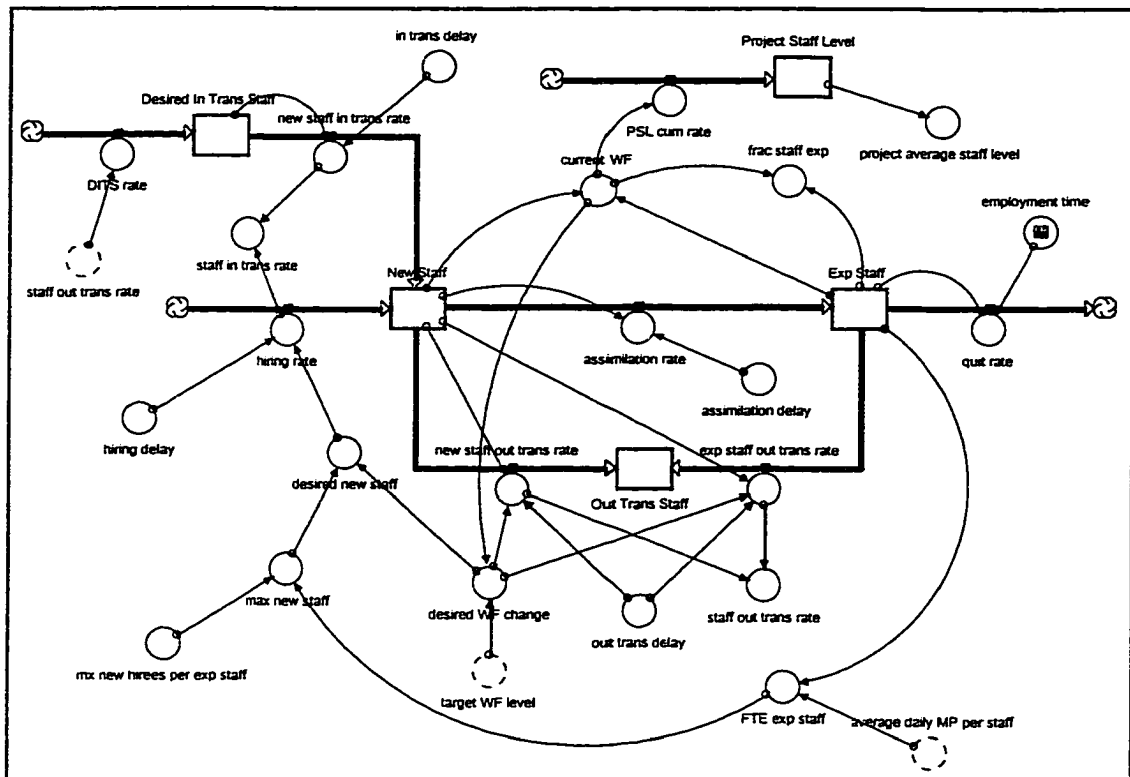


Figure A.1. The Work Force sector.

A.1.2 The Staff Productive Time Sector

The *Staff Productive Time* sector, as shown in figure A.2, monitors the staff time resource. It breaks down project staff members' daily time into two main categories: project time (*Project Time*) and slack time (*Slack Time*). Project time is the time that staff members spend on project-related activities. It is further classified into three different categories: productive time (*average productive time*), training time (*training time*), and communication time (*overall communication overhead*).

Productive time includes the time that staff members spend on development activities such as requirements specification, design, coding, testing, QA, and rework. The *training time* parameter keeps track of the time that project staff members spend in training. This includes both the time spent by experienced staff members and new staff members in training-related activities.

Communication time (*overall communication overhead*) captures the amount of time that staff members spend on communicating with other members of the project. As illustrated in figure A.3, we distinguish between communication within a team (*intrateam comm overhead*) and across teams (*interteam comm overhead*). Communication within a team usually is frequent and informal. Communication between teams usually is more formal and via meetings and/or documented agreements. A well-partitioned project usually has higher levels of communication traffic within a team than across teams.

Slack time (*Slack Time*) is the time that project staff members spend in non-project events, such as coffee breaks, personal business, and sickness. When a project is perceived to be behind schedule, people tend to work harder to bring it back on schedule. They do that by compressing their slack time and/or working overtime (*Overtime*) [7].

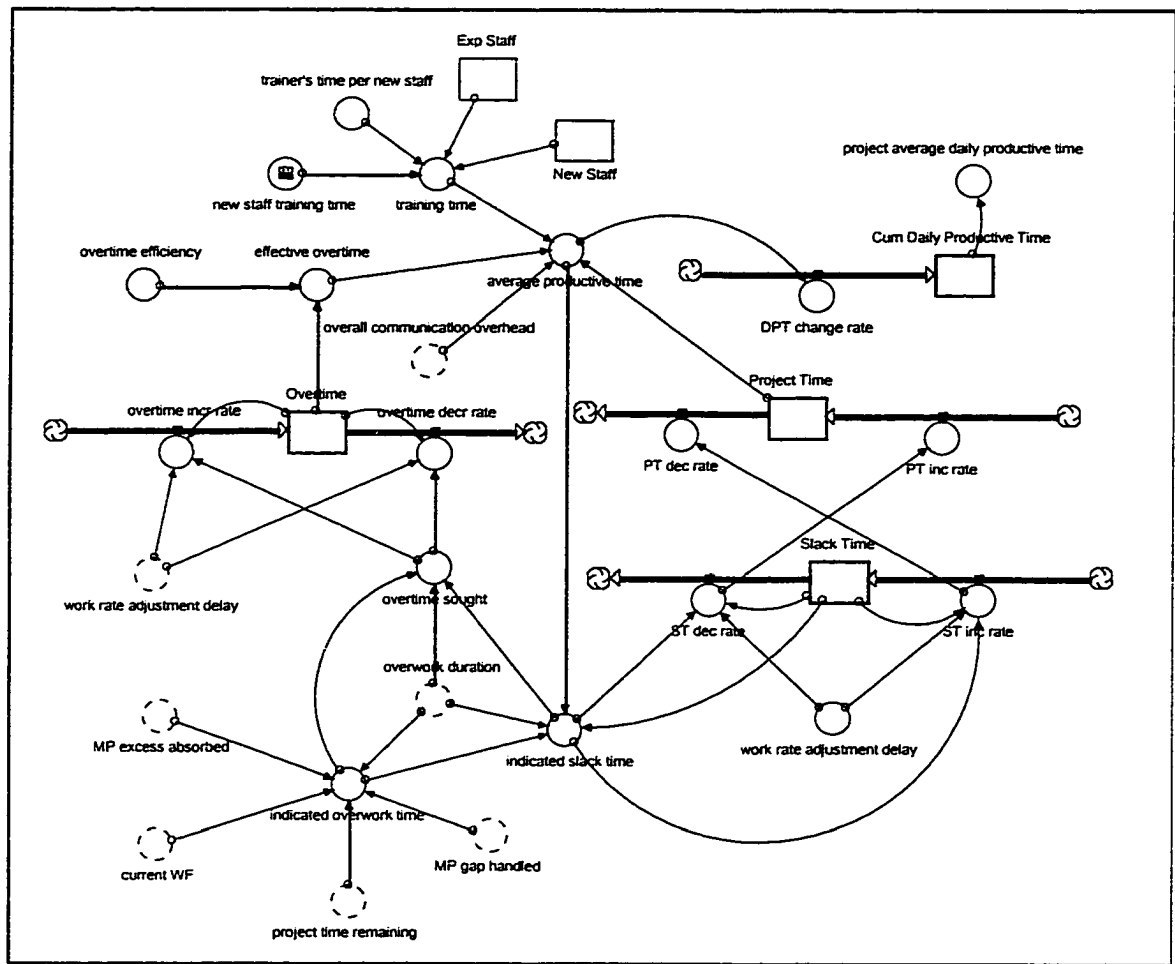


Figure A.2. The Staff Productive Time sector.

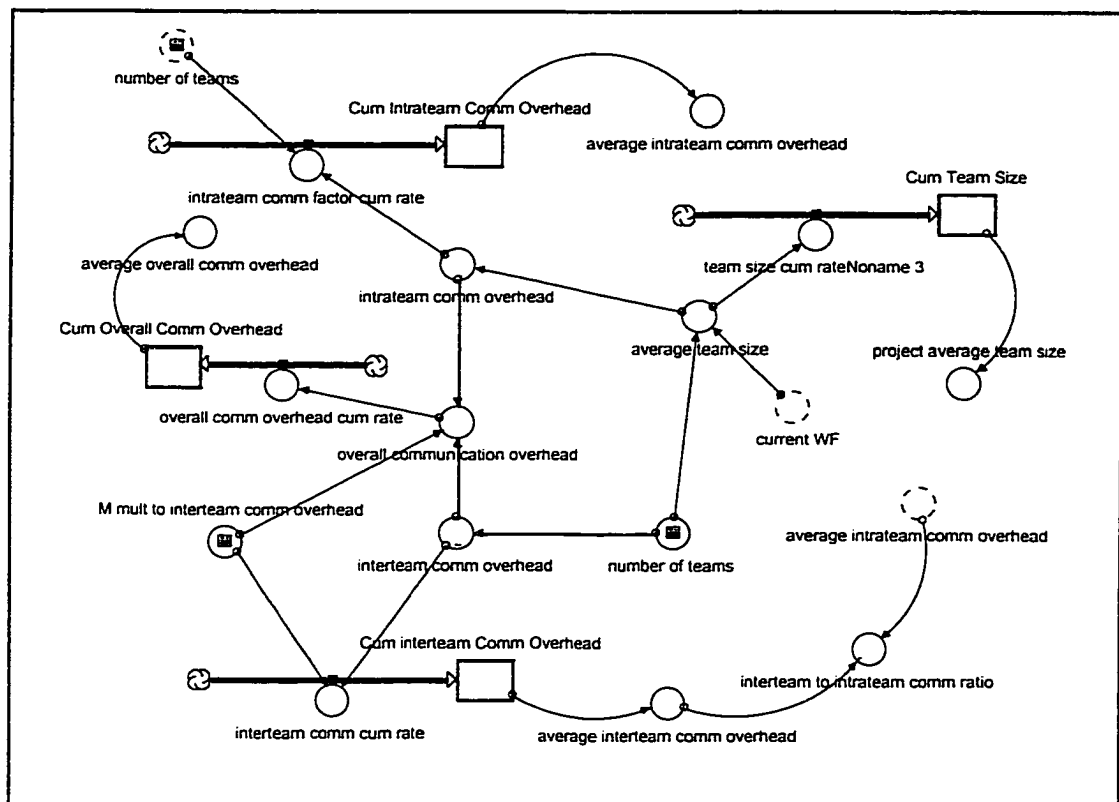


Figure A.3. Compute overall communication overhead.

A.1.3 The Staff Productivity Sector

The *Staff Productivity* sector, as illustrated in figure A.4, determines the average staff production rate, i.e., number of tasks performed per unit of time. Our focus is on project factors that likely will change over the life cycle of a software project.

In CSE-SD, staffs members' actual production rate (*actual staff prod rate*) is driven by four factors: nominal staff production rate (*nominal staff prod rate*), work force mix ratio (*frac WF exp*), schedule pressure (*schedule pressure*), and staff members' average exhaustion level (*Exhaustion Level*). The nominal staff production rate is defined as the average production rate of the experienced staff members working under the condition that there is no schedule pressure on them and they are not exhausted (i.e., *Exhaustion Level* = 0).

Exhaustion is a condition that typically results when a person works long hours across many days and takes an insufficient amount of time away from the workplace for rest and relaxation. Exhaustion can cause a person to make more mistakes, be less productive, and frequently be irritable toward coworkers [82].

In our model, exhaustion level is assumed to build up because of reduced slack time and working overtime due to schedule pressure. As the staff members continue to work overtime and/or with reduced slack time, their exhaustion level will increase. However, as their exhaustion level increases, the time span they are willing to work overtime and/or reduced slack time (*overwork duration*) decreases. At the time the exhaustion level reaches the maximum threshold exhaustion level, they are not willing to continue to overwork (i.e., *overwork duration* = 0). It will take a certain period of time without overwork (*exh diminish time*) for them to diminish the accumulated exhaustion.

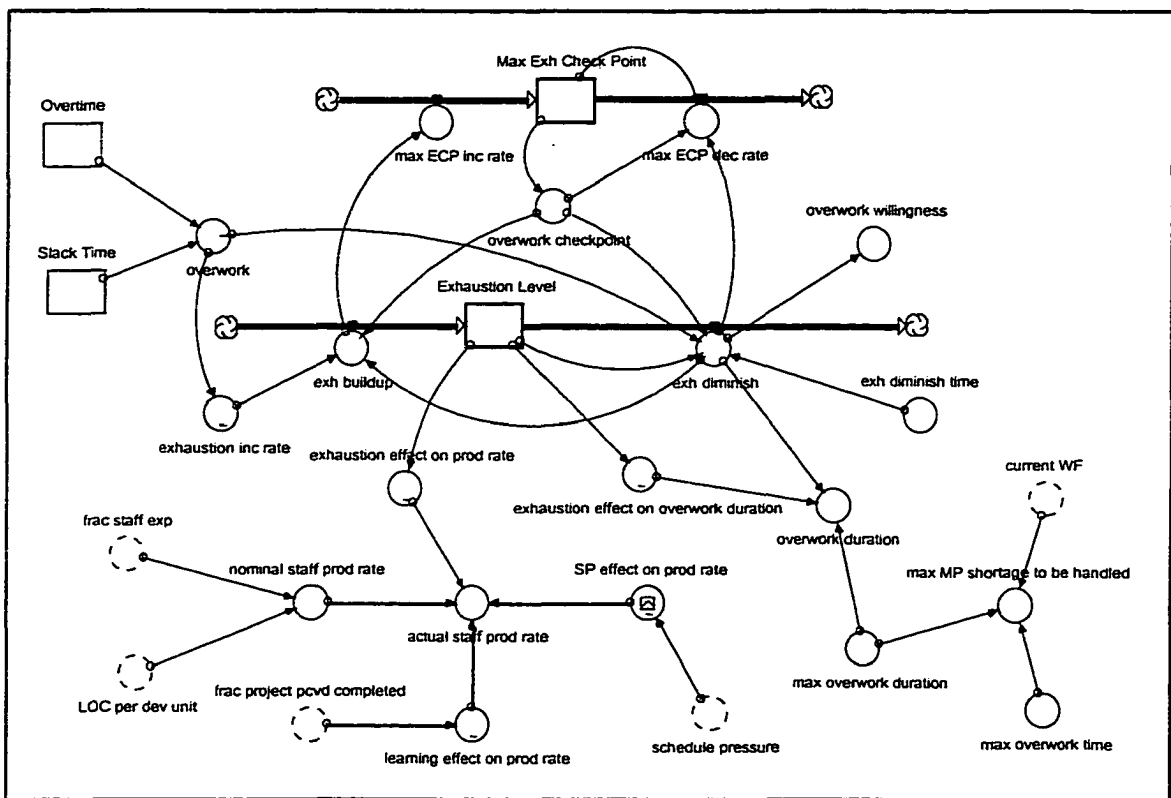


Figure A.4. The Staff Productivity sector.

A.2 The Work Flow Subsystem

The *Work Flow* subsystem models the software production activities, ranging from requirements specification to system integration and test. It consists of three sectors in which each sector models the software production process of the three phases modeled in CSE-SD, namely requirements, development, and system integration and test.

A.2.1 The Requirements Work Flow Sector

The *Requirements Work Flow* sector, as illustrated in figure A.5, models the requirements phase. Three requirements phase activities are modeled in the sector: requirements collection, requirements specification, and specification QA. The statuses of these three activities are modeled as three stock parameters: *Raw Reqs*, *Reqs Spec*, and *QAed Reqs Spec*, respectively.

The *Raw Reqs* parameter keeps track of the amount of raw requirements at any stage of the requirements phase. Despite whatever time and attention users and developers give to requirements in the beginning, they often become aware, as work proceeds, of additional features to add to the initial set of requirements [64]. The rate at which the additional requirements are incorporated into the project is modeled as the *reqs change rate* parameter (defined in the *Project Scope Change* sector).

Two sources contribute to the decrease of the *Raw Reqs* parameter. First, requirements are analyzed, and specification activity moves *Raw Reqs* into *Reqs Spec*. The speed at which *Raw Reqs* flows into *Reqs Spec* is modeled as *spec rate*, which, in turn, is determined by the amount of daily manpower allocated to requirements specification (*daily MP to spec*) and average staff requirements specification productivity (*spec prod rate*).

The second source that contributes to the decrease of raw requirements is requirements change. Requirements change may cause some of the *Raw Reqs* be deleted. The pattern of daily amount of requirements deletion due to requirements change is modeled as the *raw reqs deletion* parameter.

The *Reqs Spec* parameter keeps track of the amount of current, not-yet-QAed requirements specifications. It increases, at the rate of *spec rate*, due to the requirements specification activity. *Reqs Spec* will decrease for three reasons. First, requirements specification QA activity moves *Reqs Spec* into *QAed Reqs Spec*. The speed at which the requirements specification flows into the *QAed Reqs Spec* stock parameter is modeled as *spec QA rate*. We assume that the requirements specification QA activity follows the Parkinson's Law [22], that is, “work expands to fill the available volume.” The requirements specification QA activity will expand to use up all of the time assigned (*average QA delay*). Therefore, *spec QA rate* is modeled as *Reqs Spec* divided by *average QA delay*. The other reason that causes the requirements specification to decrease is deleted requirements specification due to requirements changes due to the discovery of unplanned requirements and the resolution of interteam interferences.

The *QAed Reqs Spec* parameter captures the amount of current QAed requirements specification. It increases, at the speed of *spec QA rate*, due to the requirements specification QA activity. Two sources cause *QAed Reqs Spec* to decrease: QAed requirements specification deleted due to requirements changes and QAed requirements specification that flow into the development phase (*QAed spec to dev rate*).

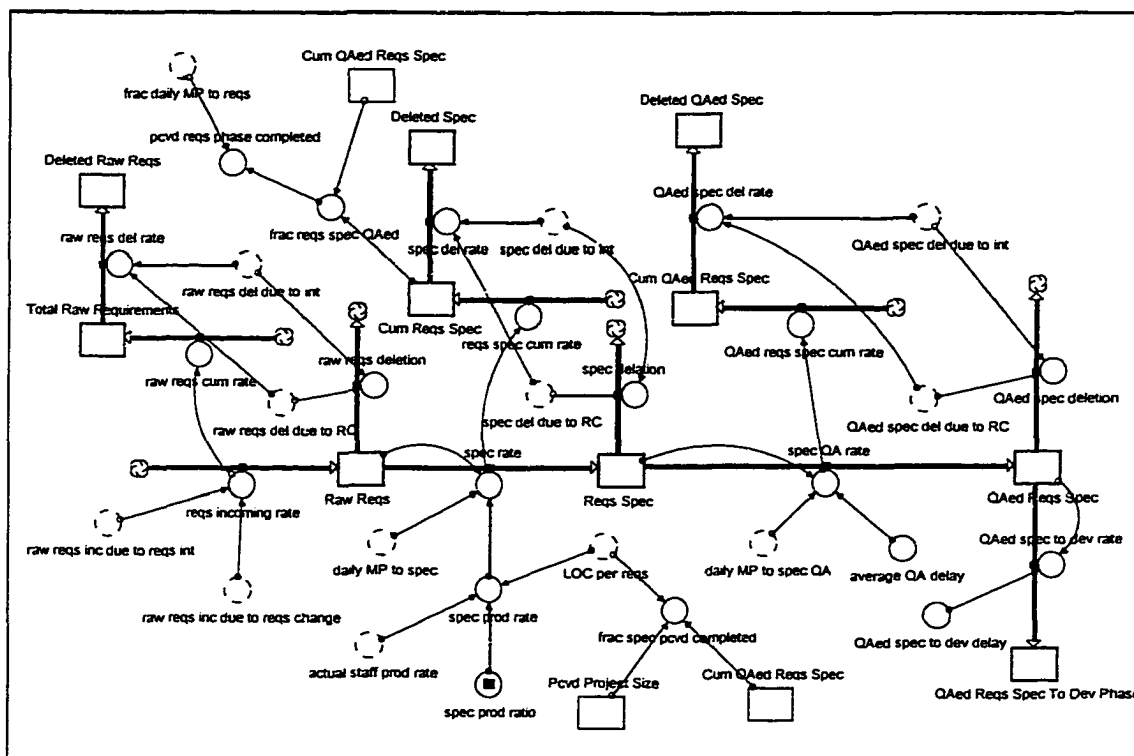


Figure A.5. The Requirements Work Flow sector.

A.2.2 The Development Work Flow Sector

As shown in figure A.6, the *Development Work Flow* sector models the development activities, including software development and QA. The QAed requirements specification coming from the *Requirements Work Flow* sector becomes the work to be performed in the development phase. The amount of work to be performed (*Units To Be Developed*) accumulates at the speed of *units TBD incoming rate*, which is defined as the sum of two parameters: *QAed spec to dev rate* and *dev units inc due to int* (development units increases due to interteam interferences).

Development phase activities are classified into two general types: development and QA. The speed at which software units are developed is modeled as the *dev rate* parameter, which is determined by the daily manpower allocated to development (*daily MP to dev*), average staff development productivity (*dev prod rate*), and degree of concurrency. Degree of concurrency is defined as the fraction of the number of software units that are ready to be worked on and the number of software units project staff members are able to perform. For example, degree of concurrency = 0.8 means that only 80% of the software units that project staff members are able to perform are ready for assignment.

The *dev QA rate* parameter models the number of developed units that are QAed per day. As with the requirements specification QA activity, we assume that the development QA follows Parkinson's Law. That is, no matter how many development units need to be QAed within a predetermined QA duration (*dev QA duration*), they always get QAed. The results of the development and development QA activities are modeled as the *Units Developed* parameter and the *Units QAed* parameter, respectively.

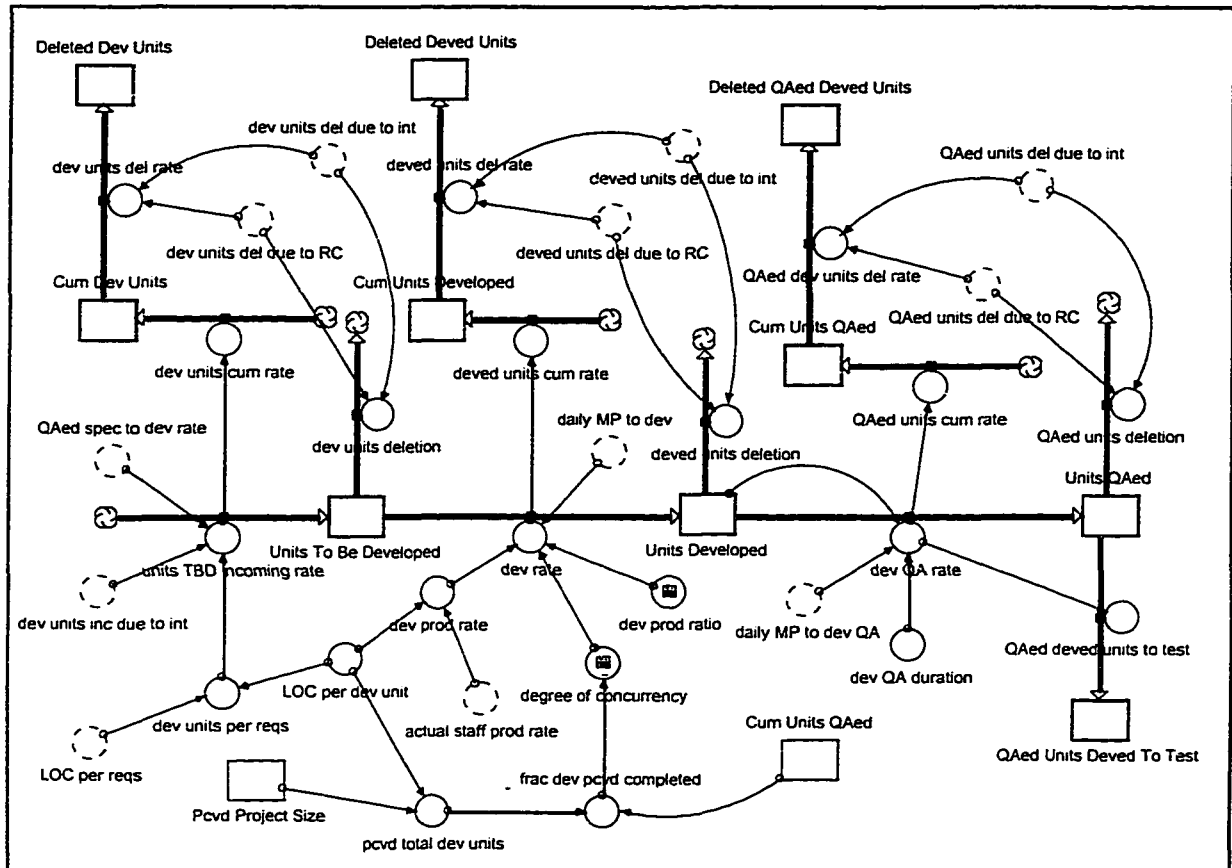


Figure A.6. The Development Work Flow sector.

A.2.3 The System Integration and Test Sector

As illustrated in figure A.7, the *System Integration and Test* sector models the system integration and test activities. Software units concurrently developed by different people and/or teams must be collected (*Units to be Integrated*) and integrated into a single system (*Units Integrated*). The integration process is a major process that serves to synchronize the multiple concurrent processes [16]. Once software units are collected, they are integrated, and then tested. The rate at which units are integrated

and tested depends on the amount of manpower allocated to system integration (*daily MP to integration*), system test (*daily MP to test*), and the average manpower to integrate and test a software unit (*testing effort per unit*).

The defects that flow into the System Integration and Test phase from the Development phase are captured in the *PreTest Defects* stock parameter. Defects are detected as the testing activity progresses. The rate at which defects are detected depends on three factors: testing rate (*testing rate*), average number of defects detected per unit tested (*num of defects detected per unit*), and test effectiveness (*test effectiveness*). Test effectiveness is defined as the fraction of defects that are detected via testing. For example, if a software unit has 10 defects, a test effectiveness of 0.8 means 8 defects will be detected when the software unit is tested. Test effectiveness is a function of daily manpower that is allocated to testing (*daily MP to test*).

Defects found in test must be corrected. The rate at which defects are corrected relies on how much manpower is allocated to correcting defects found in test (FIT) (*daily MP to defects FIT correction*) and, on average, how much effort is needed to correct a defect found in the system test (*effort to correct a defect FIT*). Defects undetected will be released to the customer (*Defects Released*).

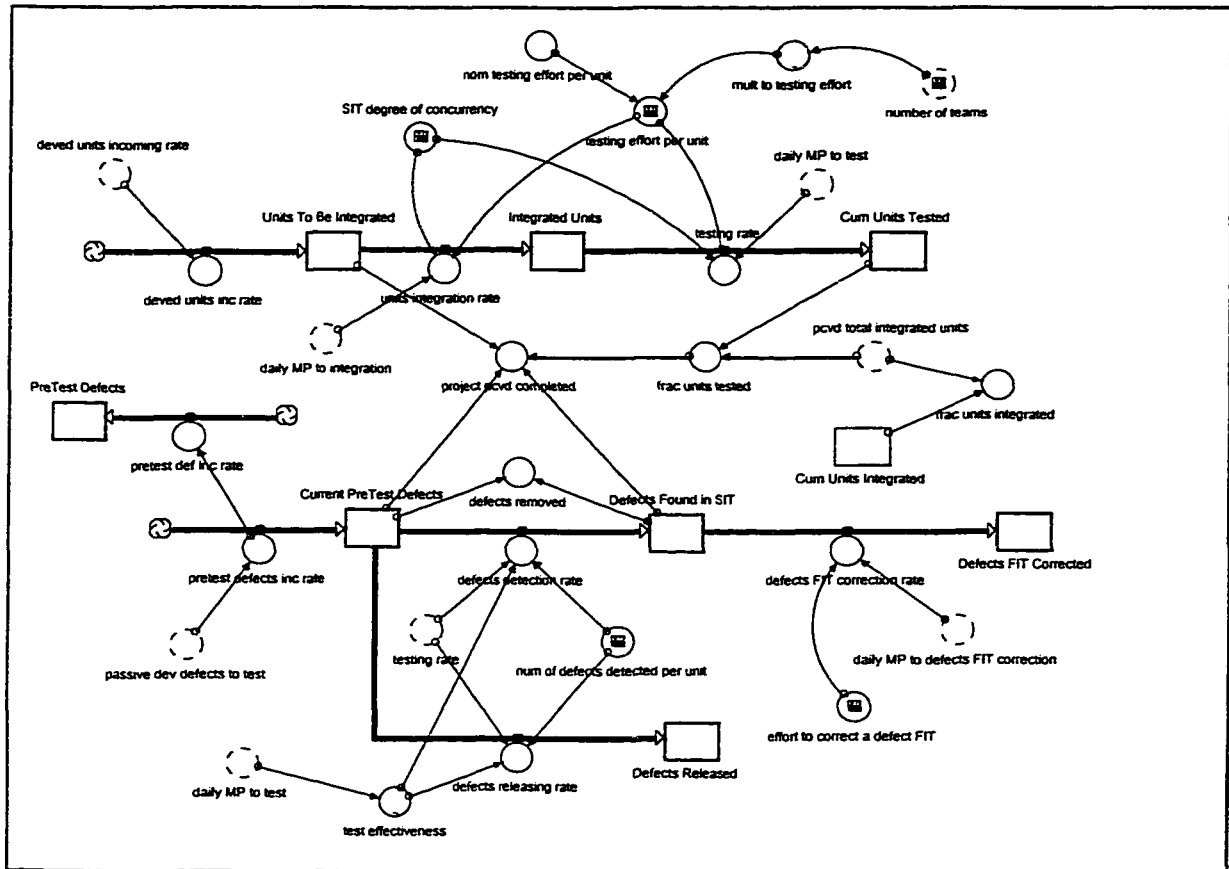


Figure A.7. The System Integration and Test sector.

A.3 The Defects and Rework Subsystem

The *Defects and Rework* subsystem models the generation, detection, and rework of detected defects. It consists of two sectors: *Requirements Defects and Rework*, and *Development Defects and Rework*. Three categories of defects are of concern: requirements defects, development defects, and bad fixes, according to the different types of activities modeled in CSE-SD. One important reason to classify defects into these three categories is that different types of defects require different

costs to fix. Defects originated in upstream phases, such as requirements, will flow into downstream phases if not detected. Designs based on defective requirements specifications are defective, no matter how perfect the design is.

A.3.3 The Requirements Defects and Rework Sector

The *Requirements Defects and Rework* sector, as illustrated in figure A.8, models the generation, detection, and correction of requirements specification defects. Requirements specifications will result in an unavoidable generation of defects. Specification defects are generated at the rate of *spec defects generation rate*, which, in turn, is determined by two parameters: the total number of requirements specified daily (*spec rate*, defined in the *Requirements Work Flow* sector) and the average number of defects generated per KLOC (*reqs defects per KLOC*).

Some of the specification defects are detected (*Detected Spec Defects*) when the specification is reviewed, and some escape detection (*Escaped Spec Defects*). Detected specification defects are then reworked (*Spec Defects Fixed*). Bad fixes to the correction of the detected specification defects (*Spec Defects Bad Fixes*) are also captured in the model. Defects that are undetected during the requirements phase and bad fixes to the detected specification defects will flow into the Development phase.

We also keep track of the density of defective requirements specification, both before (*pre QA spec defect density*) and after the QA activity (*post QA spec defect density*). Post-QA specification defect density is defined as the total number of residual specification defects (the sum of the escaped specification defects and bad fixes) divided by the cumulative number of QAed requirements specifications (*Cum QAed Reqs Spec*).

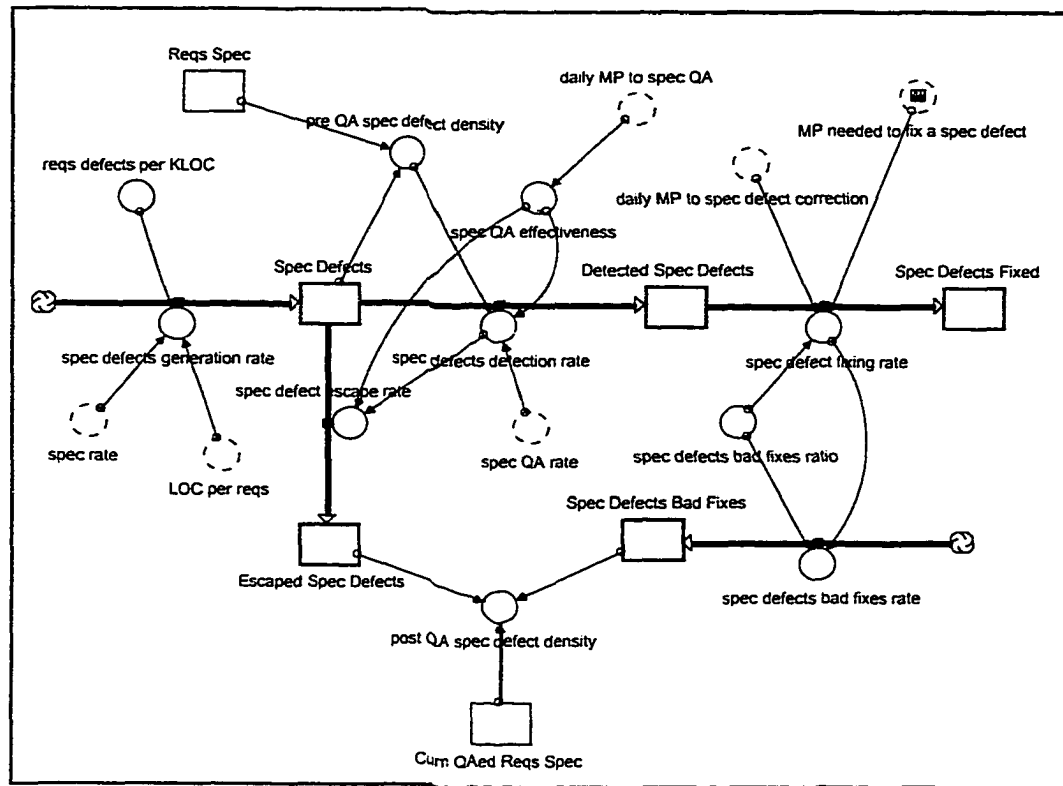


Figure A.8. The Requirements Defects and Rework sector.

A.3.4 The Development Defects and Rework Sector

The *Development Defects and Rework* sector, as shown in figure A.9, models the generation, detection, and correction of development defects, including design and coding defects. Development will result in an unavoidable generation of defects. Development defects are generated at the rate of *dev def gen rate*, which, in turn, is controlled by three parameters: (1) the number of software units developed per day (*dev rate*, as defined in the *Development Work Flow* sector); (2) the average number of

development defects committed per KLOC (*dev defects committed per KLOC*); and (3) the post-QA specification defect density (*post QA spec defect density*).

Some of the development defects are detected (*Detected Dev Defects*) when the developed software units are QAed, and some escape detection (*Cum Dev Defects Escaped*). Detected development defects are then fixed (*Cum Dev Defects Fixed*). Bad fixes to the fixing of development defects are also captured in the model (*Cum Dev Defects Bad Fixes*). Development defects that are undetected during the development phase and bad fixes to the development defects will flow into the System Integration and Test phase. Development defects that escape detection and bad fixes to the detected development defects will recycle back into the *Undetected Active Dev Defects* stock parameter.

Development defects are classified into two categories: active and passive. Active defects are defects that will amplify more defects. For example, design defects usually are active, since they will amplify coding defects. However, when the development phase progresses to the coding stage, some of the defects will not continue to amplify more defects. These passive development defects are modeled as the *Passive Dev Defects* stock parameter.

We keep track of the density of development defects (*dev defect density*), which is defined as the ratio of development defects, including both the active and passive development defects, and the cumulative software units developed (*Cum Units Deved*).

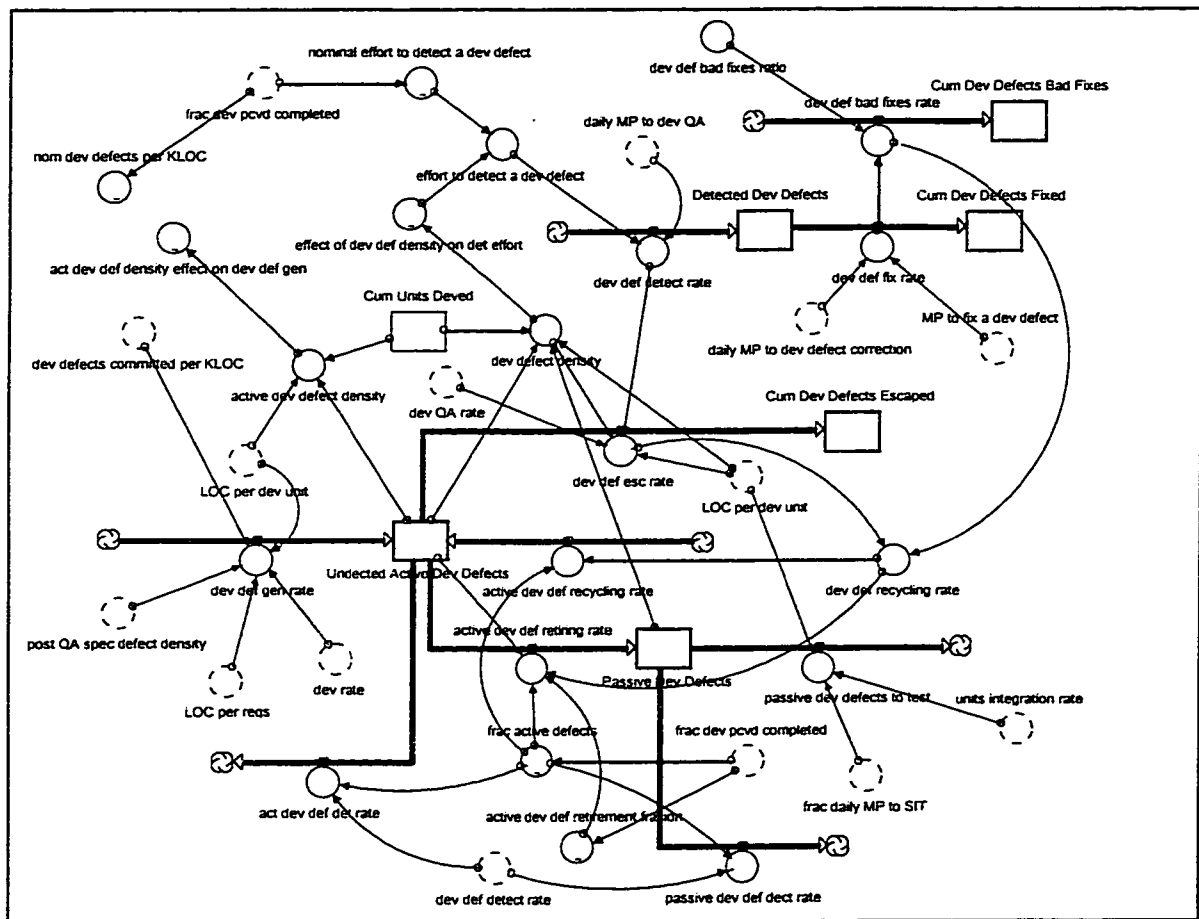


Figure A.9. The Development Defects and Rework sector.

A.4 The Manpower Allocation Subsystem

The *Manpower Allocation* subsystem allocates planned effort to different software engineering activities, including requirements specification, development, QA, defect correction, and system integration and test. It consists of three sectors: *Requirements Manpower Allocation*, *Development Manpower Allocation*, and *SIT Manpower Allocation*.

A.4.1 The Requirements Manpower Allocation Sector

The *Requirements Manpower Allocation* sector, as illustrated in figure A.10, allocates the planned daily manpower to different activities in the Requirements phase, including requirements specification (*daily MP to spec*), specification QA (*daily MP to spec QA*), requirements specification defects correction (*daily MP to spec defect correction*), requirements change rework (*daily MP to reqs change rework*), and requirements interference resolution (*daily MP to int resolution*).

Daily manpower allocated to specification defect correction is determined by two parameters: manpower needed to fix a specification defect (*MP needed to fix a spec defect*) and the desired specification defect correction rate (*desired spec defect correction rate*). The desired specification defect correction rate is determined by (1) considering the amount of detected specification defects that need to be dealt with (*Defects Spec Detected*) and (2) the average delay a specification defect is fixed after it is detected (*spec defect correction delay*).

The remaining requirements phase manpower after specification QA and specification defect correction is devoted to the requirements specification activity (*daily MP to spec*).

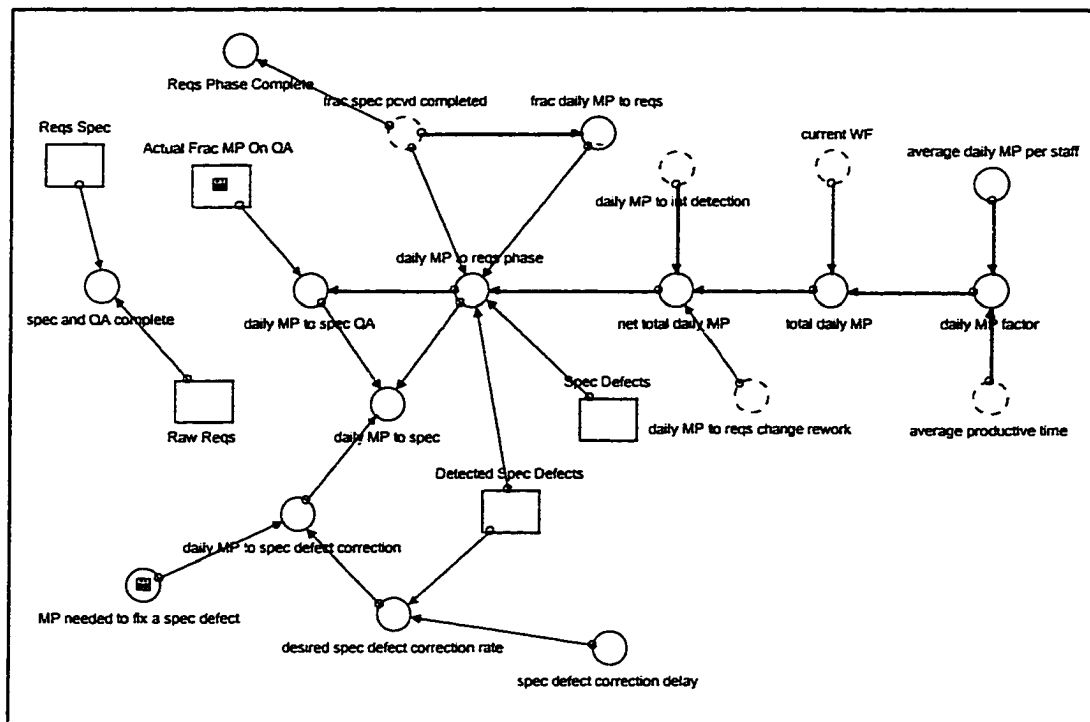


Figure A.10. The Requirements Manpower Allocation sector.

A.4.2 The Development Manpower Allocation Sector

As shown in figure A.11, the *Development Manpower Allocation* sector has a structure similar to the *Requirements Manpower Allocation* sector. Its main function is to allocate development phase manpower (*daily MP to dev phase*) to different development activities, including development, QA, and development defect correction.

Daily manpower allocated to development defect correction (*daily MP to dev defect correction*) is determined by two parameters: manpower needed to fix a development defect (*MP to fix a dev defect*) and the desired development defect correction rate (*desired dev defect correction rate*). The desired development defect correction rate

is determined by considering the amount of detected development defects that need to be fixed (*Detected Dev Defects*) and the average delay until a development defect is fixed after it is detected (*dev defect correction delay*).

The remaining development phase manpower resource after allocating to development QA (*daily MP to dev QA*) and development defect correction is allocated to the development activity (*daily MP to dev*).

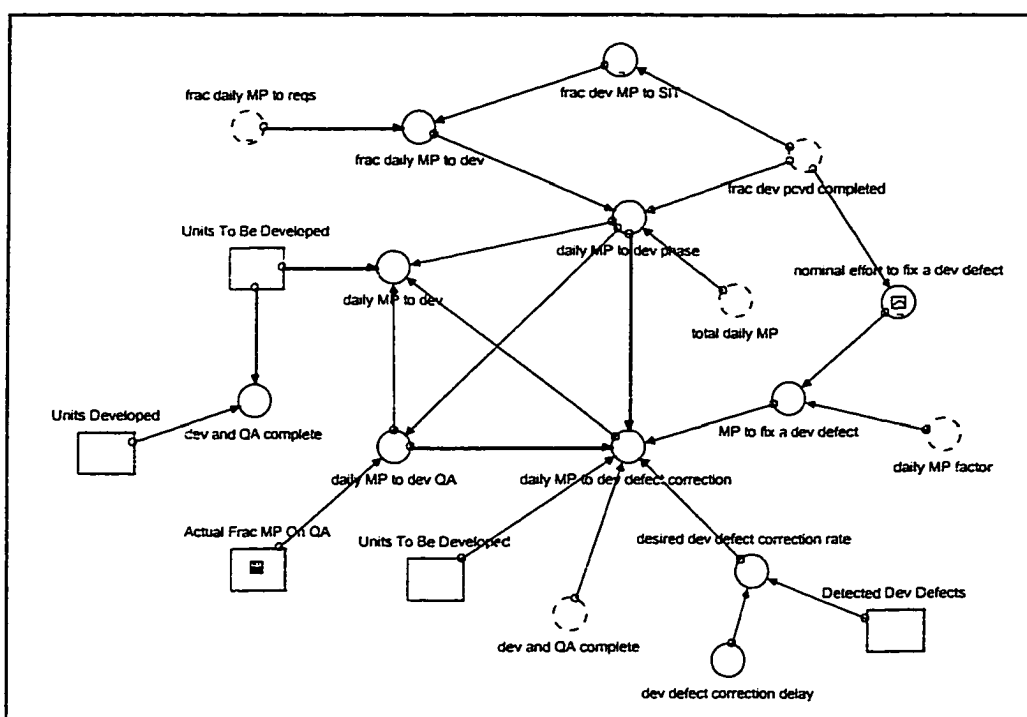


Figure A.11. The Development Manpower Allocation sector.

A.4.3 The SIT Manpower Allocation Sector

The *SIT Manpower Allocation* sector, as illustrated in figure A.12, has a structure similar to that of the *Requirements Manpower Allocation* sector and the *Development Manpower Allocation* sector. Its function is to allocate the System Integration and Test

(SIT) phase manpower (*daily MP to SIT phase*) to different activities in the SIT phase, including integration, system test, and defect correction.

Daily manpower allocated to fixing defects found in the SIT phase is determined by two parameters: manpower needed to fix a defect found in test (*MP needed to fix a defect FIT*) and the desired defect correction rate (*desired defect FIT correction rate*). The desired correction rate of the defects found in system test is determined by considering the amount of system test-detected defects that need to be corrected (*Defects Found in SIT*) and the average delay until a system test-detected defect is fixed after it is detected (*defects FIT correction delay*).

The remaining system integration and test manpower resource after allocating to system test and defect correction is allocated to the system integration activity (*daily MP to integration*).

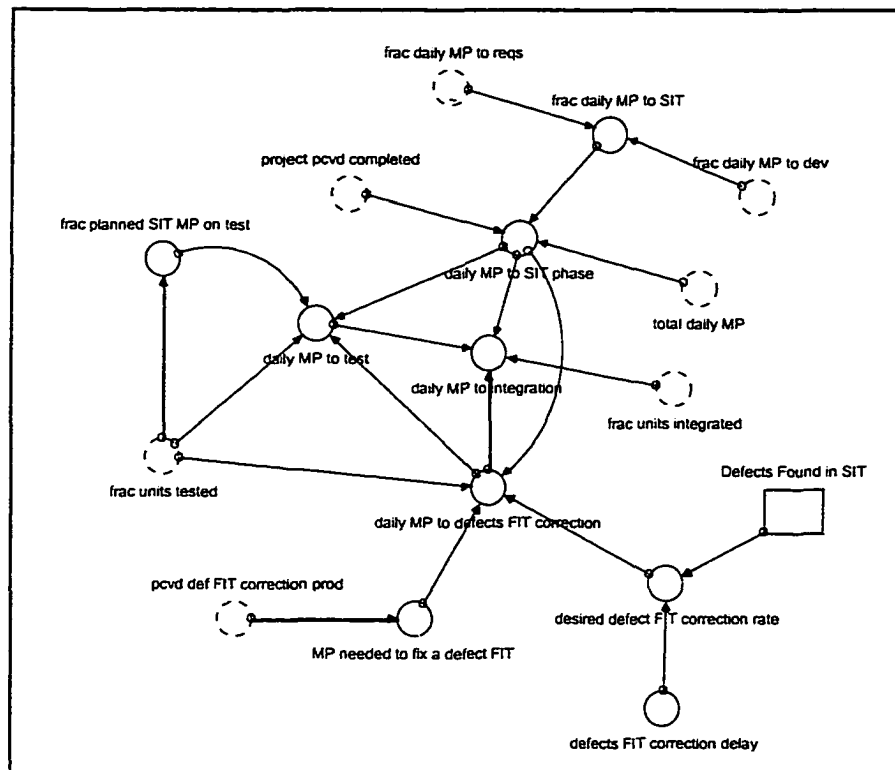


Figure A.12. The SIT Manpower Allocation sector.

A.5 The Manpower Needed Subsystem

The *Manpower Needed* subsystem determines the amount of effort perceived still needed to complete the project on time. The amount of effort perceived still needed to complete the project includes the effort perceived still needed to complete the activities in all three phases modeled in CSE-SD, namely Requirements, Development, and System Integration and Test. The effort perceived still needed to complete the Requirements, Development, and System Integration and Test phase is determined by the *Requirements Manpower Needed* sector, the *Development Manpower Needed* sector, and the *SIT Manpower Needed* sector, respectively.

A.5.5 The Requirements Manpower Needed Sector

As shown in figure A.13, the *Requirements Manpower Needed* sector determines, at any stage of the requirements phase, the effort perceived still needed to complete the requirements phase, including the effort needed for requirements specification, specification QA, and specification defect correction.

In the early stage of the requirements phase, engineers usually do not know exactly how productive they are. Their perception of their productivity simply is their planned productivity. However, when the project progresses, they begin to realize how productive they are. Therefore, their perception of their productivity approaches their actual productivity. Thus, the perception of the effort still needed to complete the requirements phase approaches the effort that is actually needed.

The perception of the manpower still needed to complete the requirements phase is modeled as a weighted average (*weight to actual reqs effort needed*) of the planned requirements phase effort remaining (*reqs phase effort remaining*) and the actual requirements phase effort needed. The actual effort still needed to complete the requirements phase is the sum of the actual specification effort needed (*actual*

spec MP needed), the specification QA effort needed (*spec QA needed*), and the specification defect correction effort needed (*spec defect correction effort needed*).

The actual effort still needed to complete the requirements specification activity is determined by dividing the total number of requirements that have been specified (*Cum Spec*) and the actual effort that was spent on the specification (*Reqs Spec Effort*). The effort that is actually needed to complete the specification activity is determined by multiplying the number of requirements remaining to be specified (*reqs remaining to be specified*) and the actual specification productivity (*actual spec productivity*).

The effort needed for specification defect correction depends on the amount of detected specification defects (*Detected Spec Defects*) and the manpower needed to fix a specification defect (*MP needed to fix a spec defect*). The actual effort still needed for specification QA is modeled as a fraction (*Actual Frac MP on QA*) of the actual specification effort perceived still needed.

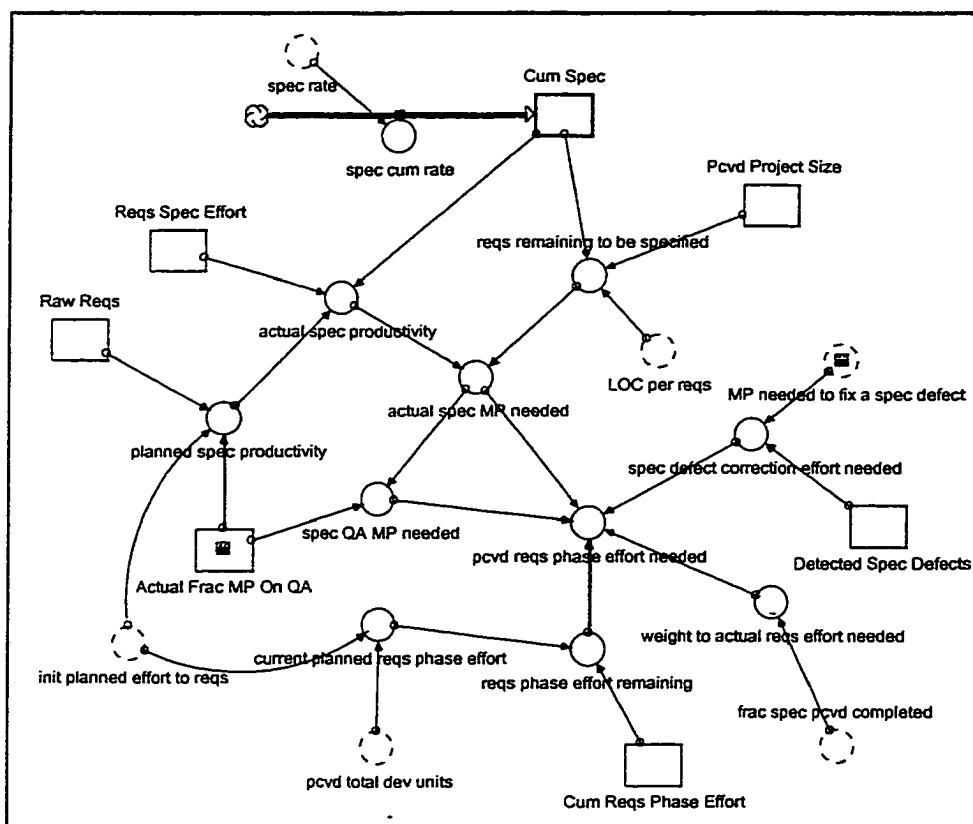


Figure A.13. The Requirements Manpower Needed sector.

A.5.6 The Development Manpower Needed Sector

As shown in figure A.14, the *Development Manpower Needed* sector determines, at any stage of the development phase, the manpower needed to complete the development phase, including manpower needed for software development, development QA, and development defect correction.

The perception of the manpower still needed to complete the development phase is modeled as a weighted average (*weight to actual dev effort needed*) of the planned development phase effort remaining (*dev phase effort remaining*) and the

actual development phase effort needed. The actual effort still needed to complete the development phase is the sum of the actual development effort needed (*actual dev effort needed*), the development QA effort needed (*dev QA MP needed*), and the development defect correction effort needed (*dev defect correction effort needed*).

The actual effort still needed to complete the development activity is determined by dividing the total number of software units that have been developed (*Cum Units Deved*) by the actual effort that was spent on it (*Cum Dev Effort*). The effort that is actually needed to complete the development activity is determined by multiplying the number of development units remaining to be developed (i.e., *pcvd total dev units - Cum Units Developed*) and the actual development production rate (*actual dev prod rate*).

The effort needed for development defect correction depends on the amount of detected development defects (*Detected Dev Defects*) and the manpower needed to fix a development defect (*MP to fix a dev defect*). The actual effort still needed for development QA is modeled as a fraction (*Actual Frac MP on QA*) of the actual development effort still needed.

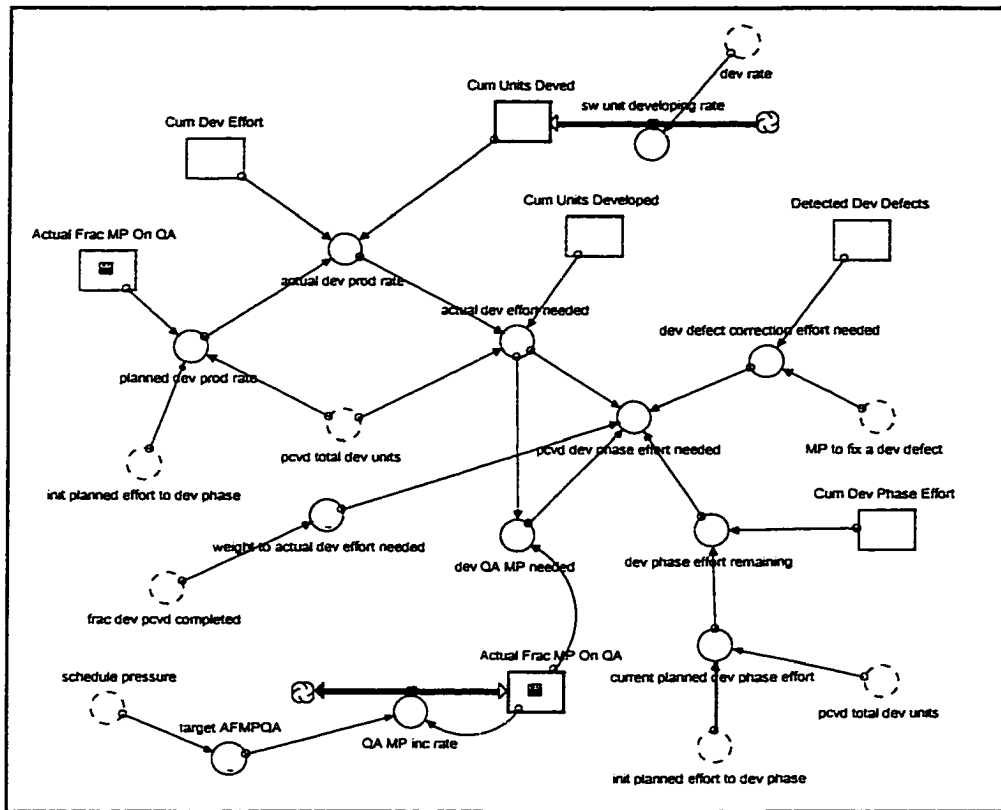


Figure A.14. The Development Manpower Needed sector.

A.5.7 The SIT Manpower Needed Sector

As shown in figure A.15, the *SIT Manpower Needed* sector determines, at any stage of the System Integration and Test (SIT) phase, the manpower needed to complete the SIT phase, including manpower needed for system integration, system test, and defects correction.

The perception of the manpower still needed to complete the SIT phase is modeled as a weighted average (*weight to actual SIT MP needed*) of the planned SIT phase effort remaining (*SIT effort remaining*) and the actual SIT phase effort needed.

The actual effort still needed to complete the SIT phase is the sum of the actual integration manpower needed (*actual itg MP needed*), the actual system test manpower needed (*actual system test MP needed*), and the defect found in the system test correction effort needed (*def FIT correction effort needed*).

The actual effort still needed to complete the system integration activity is determined by dividing the total number of development units that have been integrated (*Cum Units Integrated*) by the actual effort that was spent on it (*System Integration Effort*). The effort that is actually needed to complete the system integration activity is determined by multiplying the number of development units remaining to be integrated (i.e., *pcvd total dev units - Cum Units Integrated*) and the actual integration productivity (*actual itg prod*).

The actual effort still needed to complete the system test activity is determined by dividing the total number of integrated units that have been tested (*Cum Units Tested*) and the actual effort that was spent on it (*System Test Effort*). The effort that is actually needed to complete the system test activity is determined by multiplying the number of integrated units remaining to be tested (i.e., *pcvd total dev units - Cum Units Tested*) and the actual system test productivity (*actual system test prod*).

The actual effort needed for defects found in system test correction depends on the amount of detected defects (*Defects Found in SIT*) and the manpower needed to fix a defect found in system test (*effort to correct a defect FIT*).

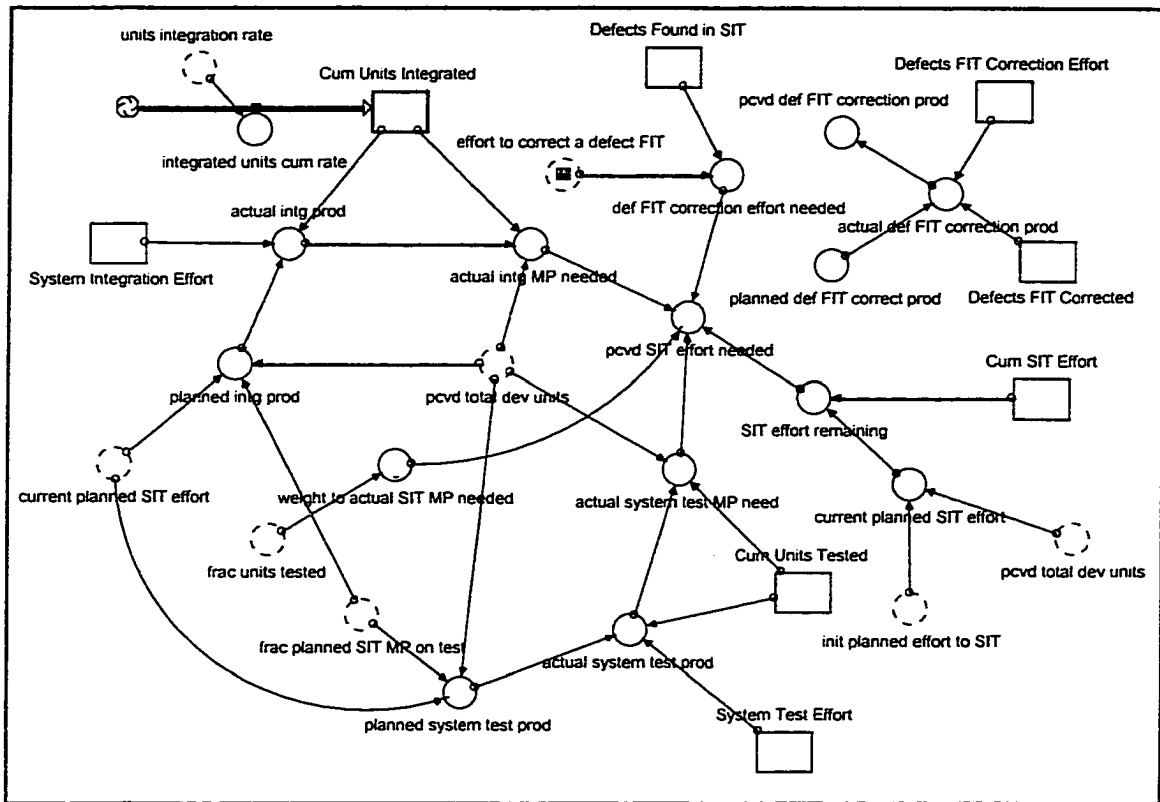


Figure A.15. The SIT Manpower Needed sector.

A.6 The Planning Sector

The *Planning* sector, as shown in figure A.16, is the entry point to the CSE-SD model. Its main functions are to compute and distribute the estimated effort, schedule, and work force to different phases of the software development life cycle. Before initiating a software development project, project managers must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved [61]. Accurate estimation of the

project effort, schedule, and required work force, however, relies on an accurate estimate of the product size.

To run the model, the simulator must provide an initial value for each of the four parameters, that is, initial estimate of the project size (*estimate of project size*), initial estimate of the required effort (*initial effort estimate*), estimated project schedule (*initial duration estimate*), and average work force (*average WF*). One also needs to determine how to distribute the planned project effort to different development phases (*pct effort to reqs*, *pct effort to dev*, and *pct effort to SIT*).

After determining the average work force (*average WF*) and the initial percentage of experienced work force (*init pct staff exp*), the initial number of experienced work force (*init exp WF*) and initial number of new work force (*init new WF*) are determined.

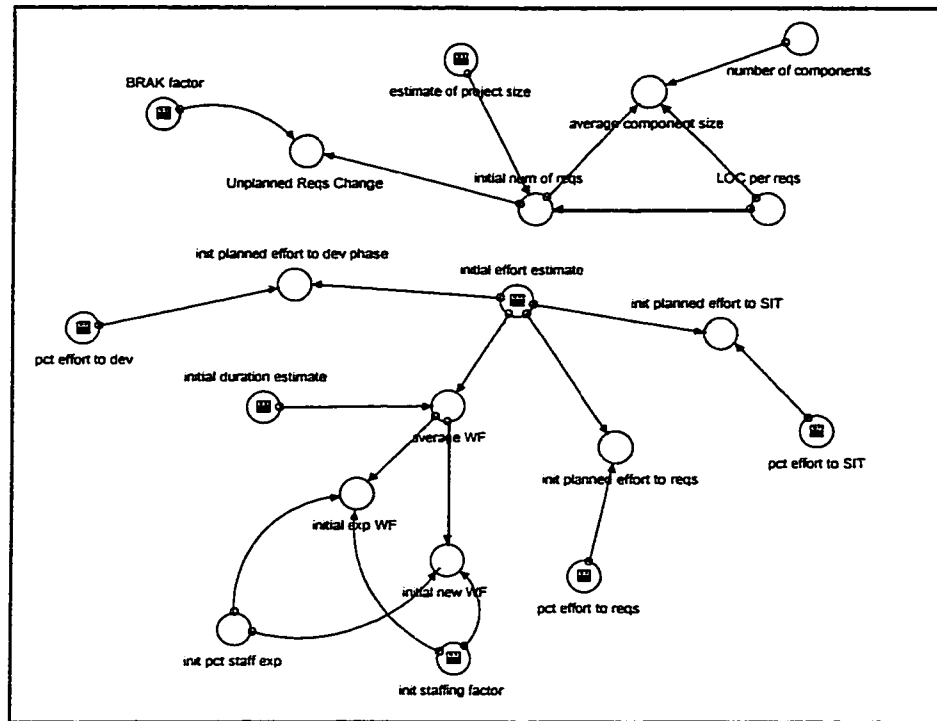


Figure A.16. The Planning sector.

A.7 The Project Control Sector

The *Project Control* sector, as shown in figure A.17, models management functions that are involved in the monitoring and control of a software development project. Monitoring is achieved by measuring and comparing the perceived software project's progress with the planned software development progress. In our model, project monitoring is achieved by comparing the project effort perceived still needed to complete the project (*pcvd project effort needed*) and the remaining planned project effort (*remaining project effort*). Ideally, if the project is on track and proceeds according to the schedule, these two measures should be identical.

If the project is perceived to be behind schedule (i.e., the perceived project effort needed exceeds remaining project effort), project staff members usually will work harder and/or work overtime trying to handle the effort gap (*MP gap handled*) and bring the project back on track. However, when the effort gap exceeds what they are able to handle, the effort gap will be reported (*project effort gap reported*).

On the other hand, if the project is perceived to be ahead of schedule (i.e., the remaining project effort exceeds the perceived project effort needed to complete the project), project staff members usually will absorb a portion of the effort excess (*MP excess absorbed*) by increasing their slack time (i.e., time spent on nonproject-related events). However, when the effort excess exceeds what they are able to absorb, the effort excess will be reported (*project effort gap reported*).

Corrective actions are taken when the project effort perceived still needed to complete the project (*pcvd project effort needed*) deviates significantly from the remaining project effort. Corrective actions that usually are taken by software project managers are modeled in CSE-SD:

1. Modify planned project effort (*Planned Project Effort*) and schedule (*Planned Project Duration*).
2. Change planned work force level (*target WF*).
3. Adjust planned QA effort, such as design review, code inspection, and testing.

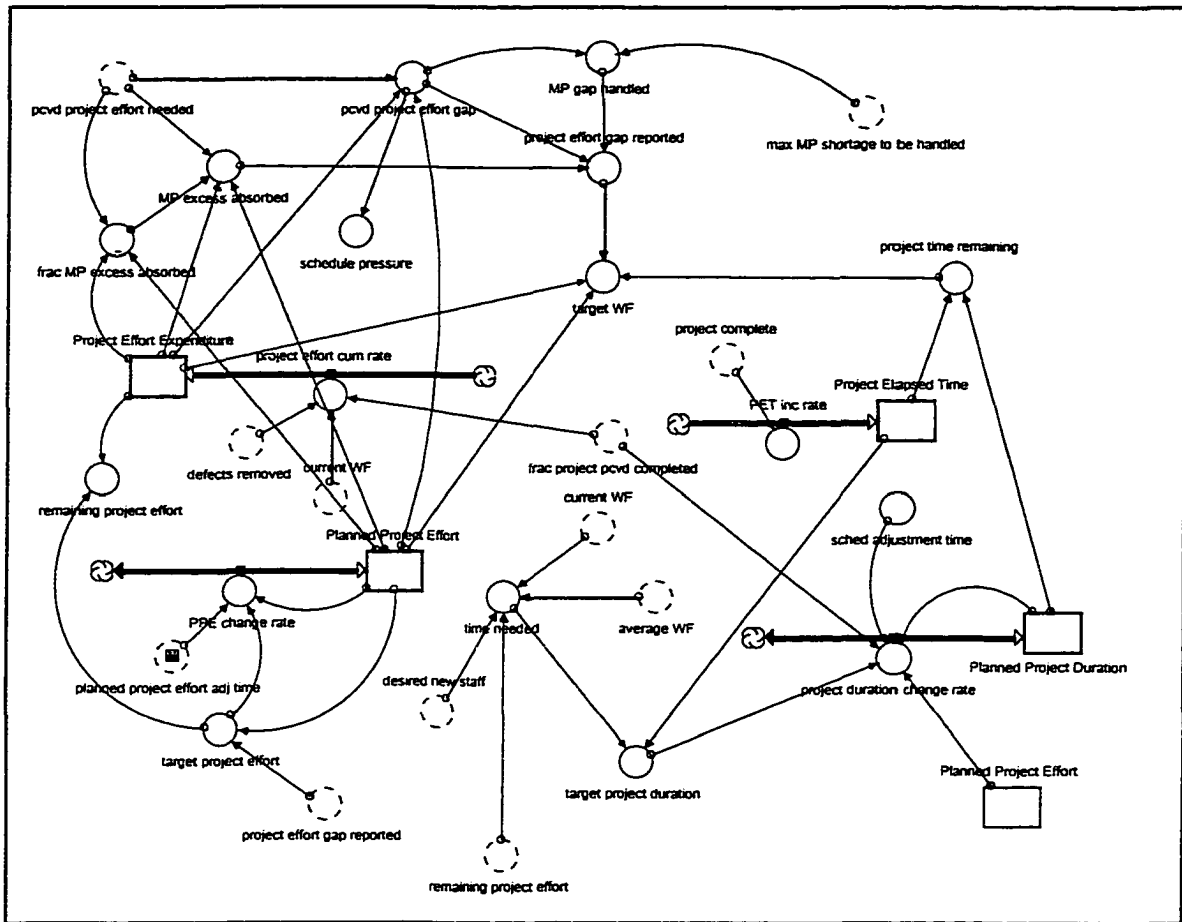


Figure A.17. The Project Control sector.

A.8 The Project Scope Change Sector

As shown in figure A.18, the *Project Scope Change* sector models the change in the scope of a software project. Reasons that cause project scope to change include incomplete and conflicting requirements specifications, requirements uncovered due to project underestimation, and new requirements. The source that causes the original project scope to change is represented as the stock parameter *Unplanned Reqs.*

The *frac unplanned reqs discovered per day* (C1, C2, and C3) parameter is defined as a function of project progress (modeled as the *frac project pcvd completed* parameter). The number of requirements discovered per day is assumed to decrease as the project progresses. Once the unplanned requirements are discovered, they are incorporated into the project plan. However, it usually takes a certain amount of time (*unplanned reqs inc delay*) before they are incorporated into the project plan. The amount of cumulative requirements changes at any stage of the development life-cycle is captured in the stock parameter *Cum Req Change*. The *reqs change rate* parameter regulates the amount of unplanned requirements incorporated into the project scope per day.

The perception of the project size (*Pcvd Project Size*) will change as unplanned requirements are discovered, existing requirements are deleted or modified, and/or new requirements are added. To simplify, we treat the modification of a requirement as a deletion and an addition of a requirement.

Requirements changes cause new raw requirements to be added and/or existing requirements (raw requirements, specification, or QAed specification) to be deleted. As depicted at the top-right portion of figure A.19, the amount of raw requirements being deleted each day (*raw reqs del due to RC*) is determined by multiplying the total number of requirements deleted each day (*reqs deletion due to RC*) by the fraction of raw requirements (*frac raw reqs*). We assume that the deleted requirements are distributed uniformly among raw requirements, specification, and QAed specification. For example, if there are six requirements to be deleted (*reqs deletion due to RC* = 6) and currently there are 10 raw requirements, 20 specifications, and 30 QAed specifications, then one raw requirements (*frac raw reqs* = 1/6), two specifications (*frac spec* = 2/6), and three QAed specifications (*frac QAed spec* = 3/6) will be deleted. The amount of specification and QAed specification being deleted each day

(*spec del due to RC* and *QAed spec del due to RC*) is determined in a similar manner. It is determined by multiplying the total number of requirements deleted each day (*reqs deletion due to RC*) and the fraction of specification (*frac spec*) by the fraction of QAed specification (*frac QAed spec*), respectively.

As illustrated in the right-bottom portion of figure A.19, the amount of development units, units developed, and QAed development units being deleted each day (*raw dev units del due to RC*, *deved units del due to RC*, and *QAed units del due to RC*) is determined in a similar manner. It is determined by multiplying the total number of development units deleted each day (*dev units deletion due to RC*) by the fraction of raw development units (*frac raw dev units*), the fraction of developed units (*frac deved units*) and the fraction of QAed units (*frac QAed units*), respectively.

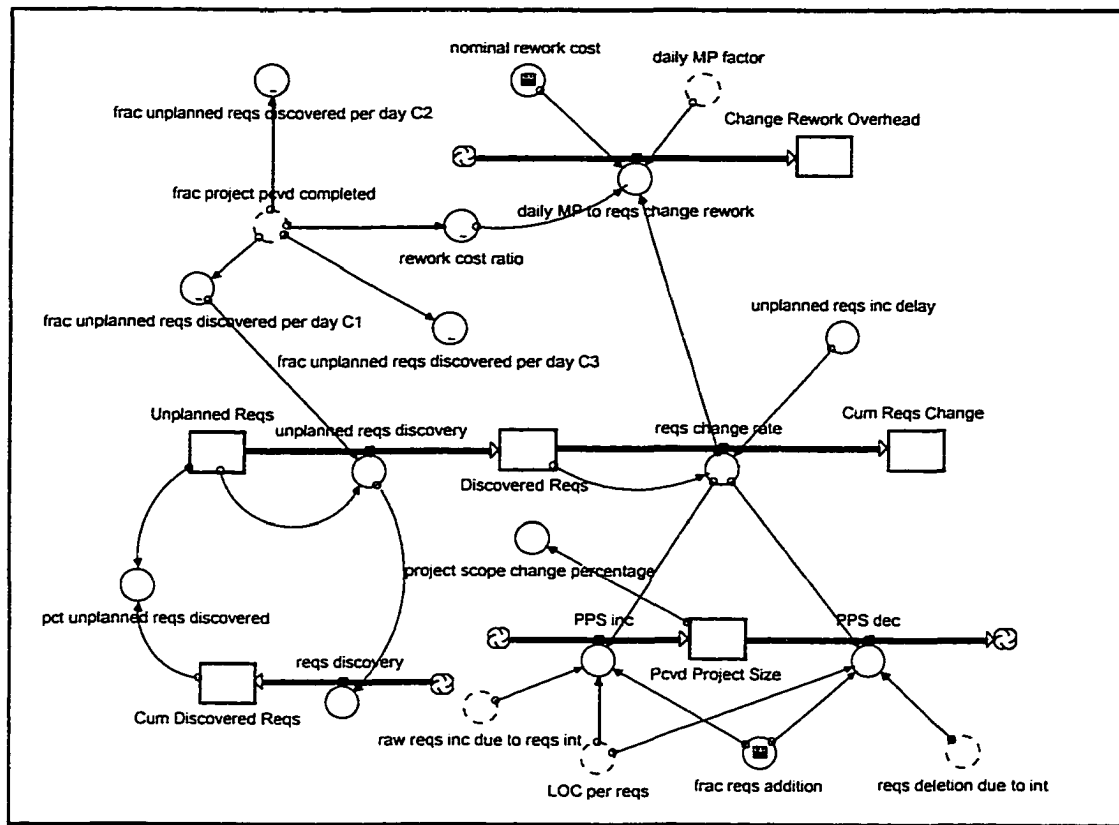


Figure A.18. The Project Scope Change sector.

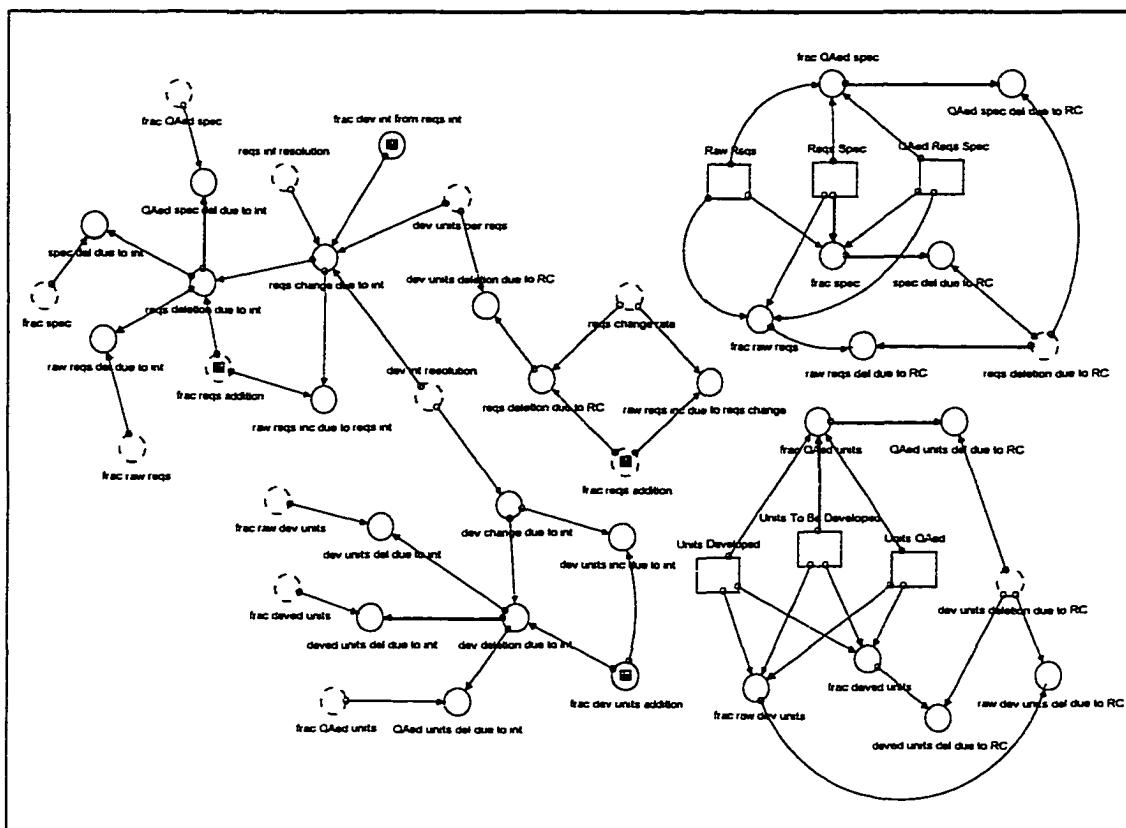


Figure A.19. Determine the amount of raw requirements, specifications, QAed specifications, development units, developed units, and QAed development units that are to be deleted due to the discovery of unplanned requirements and the resolution of interteam problems.

A.9 The Interteam Interactions Sector

As shown in figure A.20, the *Interteam Interactions* sector models the generation, detection, and resolution of problems and issues caused by multiple concurrent teams that could have been avoided if done by a single team. Multiple teams working on related subsystems may disrupt the system integrity. In requirements specifications, for example, this can cause inconsistent or incomplete specifications. In

design and implementation, simultaneous updates to a single module may violate that module's consistency [14]. Interteam problems are classified into requirements phase problems and development phase problems and are modeled as requirements interferences and development interferences, respectively.

Interferences caused by concurrent development teams are assumed to be hidden (*Undetected Reqs Ints* and *Undetected Dev Ints*) until some types of interteam synchronization and/or coordination activities are performed, for example, interteam requirements specification and design reviews. The speed at which the hidden interferences are detected is assumed to be dependent on the effort allocated to interteam issues (*daily MP to int detection*) and the average effort needed to detect an interference (*effort to detect a reqs int* and *effort to detect a dev int*).

Detected interferences of the requirements specification (*Detected Reqs Ints*) are resolved by modifying or clarifying the requirements specification. The rate at which the requirements interferences are resolved (*reqs int resolution*) is decided by how long the detected interferences are to be resolved (*int resolution delay*). That is,

$$reqs\ int\ resolution = Detected\ Reqs\ Ints / int\ resolution\ delay$$

The resolution of development interferences was modeled in a similar way.

Undetected interferences tend to propagate through succeeding tasks that build on one another, such as through design and coding tasks built on inconsistent requirement specifications. Two sources contribute to the growth of undetected development interferences (*Undetected Dev Ints*): development interference generation (*dev int gen*) and development interference regeneration (*dev int regen*). Development interference generation depends on how fast the development tasks are done (*dev rate*) and the requirements interference density (*reqs int density*), which is defined as the amount of undetected requirements interferences divided by the number of specification tasks completed. Similarly, the regeneration of development

interferences depends on the development rate and the development interference density (*dev int density*). A higher development interference density will regenerate more interferences.

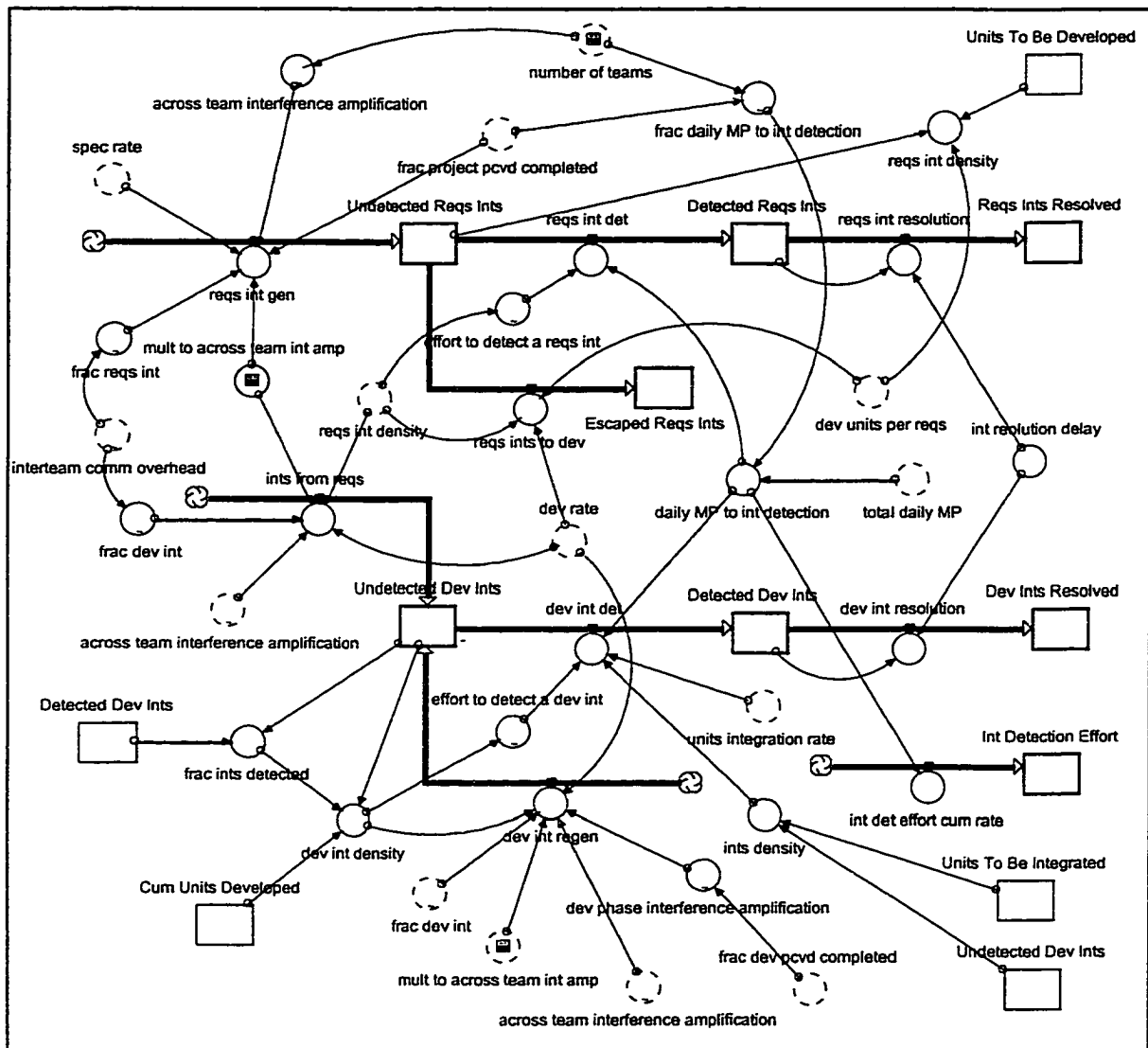


Figure A.20. The Interteam Interactions sector.

APPENDIX B
CSE-SD MODEL EQUATIONS

Determine Defect Rate

- $\text{dev_defects_committed_per_KLOC} = \text{system_complexity_effect} + \text{nom_dev_defects_per_KLOC} * (\text{WF_mix_effect_on_dev_def_gen}/1) * (\text{SP_effect_on_dev_def_gen}/1) * (\text{act_dev_def_density_effect_on_dev_def_gen}/1)$
- $\text{nom_component_size} = 100$
- $\text{nom_number_of_components} = \text{Pcvd_Project_Size} * 1000 / \text{nom_component_size}$
- $\text{SP_effect_on_dev_def_gen} = \text{GRAPH}(\text{schedule_pressure})$
(-4.00, 0.9), (-2.00, 0.94), (0.00, 1.00), (2.00, 1.05), (4.00, 1.14), (6.00, 1.24), (8.00, 1.36), (10.0, 1.50)
DOCUMENT:
Adapted from the "Multiplier to Error Generation Due to Schedule Pressure" parameter [7]
- $\text{system_complexity_effect} = \text{GRAPH}(\text{average_component_size} / \text{nom_component_size})$
(0.1, 3.50), (0.3, 1.77), (0.5, 1.00), (0.7, 0.6), (0.9, 0.75), (1.10, 0.95), (1.30, 1.10), (1.50, 1.30), (1.70, 1.60), (1.90, 1.90), (2.10, 2.20)
- $\text{WF_mix_effect_on_dev_def_gen} = \text{GRAPH}(\text{frac_staff_exp})$
(0.00, 2.00), (0.2, 1.80), (0.4, 1.60), (0.6, 1.40), (0.8, 1.20), (1.00, 1.00)
DOCUMENT:
Adapted from the "Multiplier to Error Generation Due to Workforce Mix" parameter [7]

Determine Needed Workforce

- $\text{current_time} = \text{TIME} * \text{time_scaling_factor}$
- $\text{new_planned_WF} = \text{planned_WF_20} * \text{staffing_plan_stability} + \text{target_WF} * (1 - \text{staffing_plan_stability})$
- $\text{target_WF_level} = (\text{WF_safety_factor} * \text{new_planned_WF}) * (1 - \text{WF_stability}) + \text{current_WF} * \text{WF_stability}$
- $\text{time_scaling_factor} = 633/682$
- $\text{WF_safety_factor} = 1$
- $\text{planned_WF_0} = \text{GRAPH}(\text{current_time})$
(0.00, 6.50), (50.0, 7.50), (100, 10.0), (150, 14.0), (200, 24.0), (250, 31.0), (300, 33.0), (350, 32.5), (400, 31.0), (450, 29.5), (500, 28.0), (550, 26.0), (600, 25.0), (650, 24.0)
DOCUMENT:
The planned work force distribution (BRAK = 0%)
Calibrated to produce similar work force distribution as that of COCOMO 2.0 (BRAK = 0%)
- $\text{planned_WF_10} = \text{GRAPH}(\text{current_time})$
(0.00, 6.50), (50.0, 8.00), (100, 10.0), (150, 15.0), (200, 24.0), (250, 30.0), (300, 37.0), (350, 38.0), (400, 33.0), (450, 30.0), (500, 27.0), (550, 26.0), (600, 26.0), (650, 25.0)
DOCUMENT:
The planned work force distribution (BRAK = 10%)
Calibrated to produce similar work force distribution as that of COCOMO 2.0 (BRAK = 10%)
- $\text{planned_WF_20} = \text{GRAPH}(\text{current_time})$
(0.00, 6.50), (50.0, 8.00), (100, 11.0), (150, 16.0), (200, 28.0), (250, 36.0), (300, 40.0), (350, 37.0), (400, 33.0), (450, 30.0), (500, 28.0), (550, 26.0), (600, 24.0), (650, 24.0)
DOCUMENT:
The planned workforce distribution (BRAK = 20%)
Calibrated to produce similar workforce distribution as that of COCOMO 2.0 (BRAK = 20%)

- planned_WF_25 = GRAPH(current_time)
 (0.00, 6.50), (50.0, 7.00), (100, 9.00), (150, 20.0), (200, 30.0), (250, 40.0), (300, 41.0), (350, 38.0),
 (400, 33.0), (450, 29.0), (500, 26.0), (550, 25.0), (600, 24.0), (650, 24.0)
 DOCUMENT:
 The planned workforce distribution (BRAK = 25%)
 Calibrated to produce similar workforce distribution as that of COCOMO 2.0 (BRAK = 25%)
- planned_WF_30 = GRAPH(current_time)
 (0.00, 6.50), (50.0, 7.00), (100, 9.00), (150, 20.0), (200, 30.0), (250, 40.0), (300, 41.0), (350, 39.0),
 (400, 35.0), (450, 30.0), (500, 26.0), (550, 25.0), (600, 24.0), (650, 24.0)
 DOCUMENT:
 The planned workforce distribution (BRAK = 30%)
 Calibrated to produce similar workforce distribution as that of COCOMO 2.0 (BRAK = 30%)
- planned_WF_40 = GRAPH(current_time)
 (0.00, 6.50), (50.0, 7.50), (100, 10.0), (150, 19.0), (200, 30.0), (250, 36.0), (300, 40.0), (350, 42.0),
 (400, 41.0), (450, 38.0), (500, 33.0), (550, 29.0), (600, 27.0), (650, 26.0)
 DOCUMENT:
 The planned workforce distribution (BRAK = 40%)
 Calibrated to produce similar workforce distribution as that of COCOMO 2.0 (BRAK = 40%)
- staffing_plan_stability = GRAPH(time_scaling_factor * project_time_remaining / WF_production_delay)
 (0.00, 0.00), (0.5, 0.048), (1.00, 0.138), (1.50, 0.312), (2.00, 0.582), (2.50, 0.75), (3.00, 0.87), (3.50,
 0.972), (4.00, 0.996), (4.50, 1.00), (5.00, 1.00)
- WF_stability = GRAPH(project_time_remaining / WF_production_delay)
 (0.00, 1.00), (0.3, 1.00), (0.6, 0.9), (0.9, 0.6), (1.20, 0.126), (1.50, 0.018), (1.80, 0.00)
 DOCUMENT:
 Adapted from the "Willingness to Change Work Force Level" (WCWF1) parameter [7]

Development Defects and Rework

- Cum_Dev_Defects_Bad_Fixes(t) = Cum_Dev_Defects_Bad_Fixes(t - dt) + (dev_def_bad_fixes_rate) * dt
 INIT Cum_Dev_Defects_Bad_Fixes = 0
 INFLOWS:
 dev_def_bad_fixes_rate = dev_def_fix_rate * dev_def_bad_fixes_ratio
- Cum_Dev_Defects_Escaped(t) = Cum_Dev_Defects_Escaped(t - dt) + (dev_def_esc_rate) * dt
 INIT Cum_Dev_Defects_Escaped = 0
 INFLOWS:
 dev_def_esc_rate = dev_QA_rate*(LOC_per_dev_unit/1000)*dev_defect_density - dev_def_detect_rate
- Cum_Dev_Defects_Fixed(t) = Cum_Dev_Defects_Fixed(t - dt) + (dev_def_fix_rate) * dt
 INIT Cum_Dev_Defects_Fixed = 0
 INFLOWS:
 dev_def_fix_rate = daily_MP_to_dev_defect_correction / (MP_to_fix_a_dev_defect+0.000001)
 DOCUMENT: tasks reviewed/day

- $\text{Detected_Dev_Defects}(t) = \text{Detected_Dev_Defects}(t - dt) + (\text{dev_def_detect_rate} - \text{dev_def_fix_rate}) * dt$
 INIT $\text{Detected_Dev_Defects} = 0$
 INFLOWS:
 ☞ $\text{dev_def_detect_rate} = \text{daily_MP_to_dev_QA} / (\text{effort_to_detect_a_dev_defect}/1)$
 OUTFLOWS:
 ☞ $\text{dev_def_fix_rate} = \text{daily_MP_to_dev_defect_correction} / (\text{MP_to_fix_a_dev_defect} + 0.000001)$
 DOCUMENT: tasks reviewed/day
- $\text{Passive_Dev_Defects}(t) = \text{Passive_Dev_Defects}(t - dt) + (\text{active_dev_def_retiring_rate} - \text{passive_dev_defects_to_test} - \text{passive_dev_def_det_rate}) * dt$
 INIT $\text{Passive_Dev_Defects} = 0$
 INFLOWS:
 ☞ $\text{active_dev_def_retiring_rate} = \text{Undected_Active_Dev_Defects} * \text{active_dev_def_retirement_fraction} + \text{dev_def_recycling_rate} * (1 - \text{frac_active_defects})$
 OUTFLOWS:
 ☞ $\text{passive_dev_defects_to_test} = \text{IF}(\text{frac_daily_MP_to_SIT} > 0) \text{ THEN } (\text{units_integration_rate} * \text{LOC_per_dev_unit}/1000) * 23$
 ELSE 0
 DOCUMENT: 27 defects/unit integrated
 ☞ $\text{passive_dev_def_det_rate} = \text{dev_def_detect_rate} * (1 - \text{frac_active_defects})$
- $\text{Undected_Active_Dev_Defects}(t) = \text{Undected_Active_Dev_Defects}(t - dt) + (\text{dev_def_gen_rate} + \text{active_dev_def_recycling_rate} - \text{dev_def_esc_rate} - \text{active_dev_def_retiring_rate} - \text{act_dev_def_det_rate}) * dt$
 INIT $\text{Undected_Active_Dev_Defects} = 0$
 INFLOWS:
 ☞ $\text{dev_def_gen_rate} = (1 - \text{post_QA_spec_defect_density}) * (\text{dev_rate} * \text{LOC_per_dev_unit}/1000) * \text{dev_defects_committed_per_KLOC} +$
 $0 * (\text{post_QA_spec_defect_density} * \text{dev_rate} * \text{LOC_per_reqs}/1000) * \text{dev_defects_committed_per_KLOC}$
 ☞ $\text{active_dev_def_recycling_rate} = \text{dev_def_recycling_rate} * \text{frac_active_defects}$
- OUTFLOWS:
 ☞ $\text{dev_def_esc_rate} = \text{dev_QA_rate} * (\text{LOC_per_dev_unit}/1000) * \text{dev_defect_density} - \text{dev_def_detect_rate}$
 ☞ $\text{active_dev_def_retiring_rate} = \text{Undected_Active_Dev_Defects} * \text{active_dev_def_retirement_fraction} + \text{dev_def_recycling_rate} * (1 - \text{frac_active_defects})$
 ☞ $\text{act_dev_def_det_rate} = \text{dev_def_detect_rate} * \text{frac_active_defects}$
- $\text{active_dev_defect_density} = \text{Undected_Active_Dev_Defects} / (\text{Cum_Units_Deved} * \text{LOC_per_dev_unit}/1000 + 0.00001)$
- $\text{dev_defect_density} = ((\text{Undected_Active_Dev_Defects} + \text{Passive_Dev_Defects})) / (\text{Cum_Units_Deved} * \text{LOC_per_dev_unit}/1000 + 0.00001)$

- `dev_def_bad_fixes_ratio = 0.075`
DOCUMENT:
Development (design and coding) defects bad fixes ratio
Set to 0.075 [Jones 91]
- `dev_def_recycling_rate = dev_def_esc_rate + dev_def_bad_fixes_rate`
- `effort_to_detect_a_dev_defect = nominal_effort_to_detect_a_dev_defect *
effect_of_dev_def_density_on_det_effort`
- `active_dev_def_retirement_fraction = GRAPH(frac_dev_pcvd_completed)`
(0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.00), (0.4, 0.01), (0.5, 0.02), (0.6, 0.03), (0.7, 0.04), (0.8, 0.1), (0.9, 0.3), (1, 1.00)
DOCUMENT:
Adapted from the "Active Error Retirement Rate" parameter [7]
- `act_dev_def_density_effect_on_dev_def_gen = GRAPH(SMTH1(active_dev_defect_density, 40))`
(0.00, 1.00), (10.0, 1.10), (20.0, 1.20), (30.0, 1.33), (40.0, 1.45), (50.0, 1.60), (60.0, 2.00), (70.0, 2.50), (80.0, 3.25), (90.0, 4.35), (100, 6.00)
DOCUMENT:
The delay before one defect amplifies additional defects
The average delay is set at two months (40 working days) [7]
- `effect_of_dev_def_density_on_det_effort = GRAPH(dev_defect_density)`
(0.00, 50.0), (1.00, 36.0), (2.00, 26.0), (3.00, 17.5), (4.00, 10.0), (5.00, 4.00), (6.00, 1.75), (7.00, 1.20), (8.00, 1.00), (9.00, 1.00), (10.0, 1.00)
- `frac_active_defects = GRAPH(frac_dev_pcvd_completed)`
(0.00, 1.00), (0.1, 1.00), (0.2, 1.00), (0.3, 1.00), (0.4, 0.95), (0.5, 0.85), (0.6, 0.5), (0.7, 0.2), (0.8, 0.075), (0.9, 0.00), (1, 0.00)
DOCUMENT:
The percentage of active defects is defined as a graph function of percent of development completed
Adapted from the "Percent Active Errors" parameter [7]
- `nominal_effort_to_detect_a_dev_defect = GRAPH(frac_dev_pcvd_completed)`
(0.00, 0.4), (0.1, 0.4), (0.2, 0.39), (0.3, 0.375), (0.4, 0.35), (0.5, 0.3), (0.6, 0.25), (0.7, 0.225), (0.8, 0.21), (0.9, 0.2), (1, 0.2)
DOCUMENT:
Average QA effort needed to detect a development (including design and coding) defect
Adapted from the "Nominal QA Effort Needed to Detect an Error" parameter [7]
- `nom_dev_defects_per_KLOC = GRAPH(frac_dev_pcvd_completed)`
(0.00, 25.0), (0.2, 23.9), (0.4, 21.6), (0.6, 15.9), (0.8, 13.6), (1.00, 12.5)
DOCUMENT:
Nominal development (design and coding) defects committed per KLOC
Adapted from the "Nominal Number of Errors Committed per KDSI" parameter [7]

Development Manpower Allocation

- `daily_MP_to_dev = IF(Units_To_Be_Developed<0.1)
THEN 0
ELSE
MAX(daily_MP_to_dev_phase-daily_MP_to_dev_QA-daily_MP_to_dev_defect_correction,0)`

- $\text{daily_MP_to_dev_defect_correction} = \text{IF}(\text{dev_and_QA_complete}=1)$
 THEN $\text{daily_MP_to_dev_phase}$
 ELSE $\text{IF}(\text{Units_To_Be_Developed}<0.1)$
 THEN $\text{daily_MP_to_dev_phase} - \text{daily_MP_to_dev_QA}$
 ELSE $\text{MIN}(\text{MP_to_fix_a_dev_defect} * \text{desired_dev_defect_correction_rate},$
 $\text{daily_MP_to_dev_phase} - \text{daily_MP_to_dev_QA})$
- $\text{daily_MP_to_dev_phase} = \text{IF}(\text{frac_dev_pcvd_completed}=1)$
 THEN 0
 ELSE
 $\text{total_daily_MP} * \text{frac_daily_MP_to_dev}$
- $\text{daily_MP_to_dev_QA} = \text{daily_MP_to_dev_phase} * \text{Actual_Frac_MP_On_QA}$
- $\text{desired_dev_defect_correction_rate} = \text{Detected_Dev_Defects} / \text{dev_defect_correction_delay}$
- $\text{dev_and_QA_complete} = \text{IF}(\text{Units_To_Be_Developed}<1 \text{ AND } \text{Units_Developed}<1)$
 THEN 1
 ELSE 0
- $\text{dev_defect_correction_delay} = 15$
 DOCUMENT:
 Set to 15 working days
 Similar to the "Desired Rework Delay" parameter [7]
- $\text{frac_daily_MP_to_dev} = (1 - \text{frac_daily_MP_to_reqs}) * (1 - \text{frac_dev_MP_to_SIT})$
- $\text{MP_to_fix_a_dev_defect} = \text{daily_MP_factor} * \text{nominal_effort_to_fix_a_dev_defect}$
- $\text{frac_dev_MP_to_SIT} = \text{GRAPH}(\text{frac_dev_pcvd_completed})$
 (0.5, 0.00), (0.55, 0.00), (0.6, 0.00), (0.65, 0.00), (0.7, 0.00), (0.75, 0.00), (0.8, 0.00), (0.85, 0.00), (0.9, 0.00), (0.95, 0.00), (1.00, 1.00)
 DOCUMENT:
 Determined by project managers to simulate different manpower allocation policy
- $\text{nominal_effort_to_fix_a_dev_defect} = \text{GRAPH}(\text{frac_dev_pcvd_completed})$
 (0.00, 0.6), (0.2, 0.575), (0.4, 0.5), (0.6, 0.4), (0.8, 0.325), (1.00, 0.3)
 DOCUMENT:
 Nominal defect correction effort
 Adapted from the "Nominal Rework Effort Needed per Error" parameter [7]

Development Manpower Needed

- $\text{Actual_Frac_MP_On_QA}(t) = \text{Actual_Frac_MP_On_QA}(t - dt) + (\text{QA_MP_inc_rate}) * dt$
 INIT $\text{Actual_Frac_MP_On_QA} = 0.1$
 INFLOWS:
 $\text{QA_MP_inc_rate} = 0 * (\text{target_AFMPQA} - \text{Actual_Frac_MP_On_QA}) / 1$
- $\text{Cum_Units_Deved}(t) = \text{Cum_Units_Deved}(t - dt) + (\text{sw_unit_developing_rate}) * dt$
 INIT $\text{Cum_Units_Deved} = 0$
 INFLOWS:
 $\text{sw_unit_developing_rate} = \text{dev_rate}$
- $\text{actual_dev_effort_needed} = (\text{pcvd_total_dev_units} - \text{Cum_Units_Developed}) /$
 $(\text{actual_dev_prod_rate} + 0.000001)$

- $actual_dev_prod_rate = IF(Cum_Dev_Effort > 0)$
 $THEN Cum_Units_Deved / (Cum_Dev_Effort + 0.000001)$
 $ELSE planned_dev_prod_rate$
- $current_planned_dev_phase_effort = init_planned_effort_to_dev_phase * (pcvd_total_dev_units / INIT(pcvd_total_dev_units))$
- $dev_defect_correction_effort_needed = Detected_Dev_Defects * MP_to_fix_a_dev_defect$
- $dev_phase_effort_remaining = MAX(0, current_planned_dev_phase_effort - Cum_Dev_Phase_Effort)$
- $dev_QA_MP_needed = (actual_dev_effort_needed / (1 - Actual_Frac_MP_On_QA)) * Actual_Frac_MP_On_QA$
- $pcvd_dev_phase_effort_needed = weight_to_actual_dev_effort_needed * (actual_dev_effort_needed + dev_defect_correction_effort_needed + dev_QA_MP_needed) + (1 - weight_to_actual_dev_effort_needed) * dev_phase_effort_remaining$
- $planned_dev_prod_rate = pcvd_total_dev_units / (init_planned_effort_to_dev_phase * (1 - Actual_Frac_MP_On_QA))$
 DOCUMENT:
 The planned development production rate
 Perceived total number of development units divided by the planned development effort
- $target_AFMPQA = GRAPH(schedule_pressure)$
 $(0.00, 0.15), (1.00, 0.15), (2.00, 0.15), (3.00, 0.15), (4.00, 0.15), (5.00, 0.145), (6.00, 0.131), (7.00, 0.102), (8.00, 0.071), (9.00, 0.055), (10.0, 0.05)$
 DOCUMENT:
 The effect of schedule pressure on QA manpower allocation
 Adapted from the "Planned Fraction of Manpower for QA" parameter [7]
- $weight_to_actual_dev_effort_needed = GRAPH(frac_dev_pcvd_completed)$
 $(0.00, 0.00), (0.1, 0.01), (0.2, 0.05), (0.3, 0.174), (0.4, 0.432), (0.5, 0.714), (0.6, 0.858), (0.7, 0.936), (0.8, 0.984), (0.9, 0.996), (1, 1.00)$
 DOCUMENT:
 Adapted from the "Multiplier to Productivity Weight Due to Resource Expenditures" and the "Multiplier to Productivity Weight Due to Development" parameters [7]

Development Work Flow

- $Cum_Dev_Units(t) = Cum_Dev_Units(t - dt) + (dev_units_cum_rate - dev_units_del_rate) * dt$
 $INIT Cum_Dev_Units = 0$
 INFLOWS:
 $dev_units_cum_rate = units_TBD_incoming_rate$
 OUTFLOWS:
 $dev_units_del_rate = raw_dev_units_del_due_to_RC + dev_units_del_due_to_int$
- $Cum_Units_Developed(t) = Cum_Units_Developed(t - dt) + (deved_units_cum_rate - deved_units_del_rate) * dt$
 $INIT Cum_Units_Developed = 0$
 INFLOWS:
 $deved_units_cum_rate = dev_rate$
 OUTFLOWS:

- \Rightarrow $deved_units_del_rate = deved_units_del_due_to_RC + deved_units_del_due_to_int$
 $Cum_Units_QAed(t) = Cum_Units_QAed(t - dt) + (QAed_units_cum_rate - QAed_dev_units_del_rate) * dt$
 INIT $Cum_Units_QAed = 0$
 INFLOWS:
 \Rightarrow $QAed_units_cum_rate = dev_QA_rate$
 OUTFLOWS:
 \Rightarrow $QAed_dev_units_del_rate = QAed_units_del_due_to_RC + QAed_units_del_due_to_int$
 $Deleted_Deved_Units(t) = Deleted_Deved_Units(t - dt) + (deved_units_del_rate) * dt$
 INIT $Deleted_Deved_Units = 0$
 INFLOWS:
 \Rightarrow $deved_units_del_rate = deved_units_del_due_to_RC + deved_units_del_due_to_int$
 $Deleted_Dev_Units(t) = Deleted_Dev_Units(t - dt) + (dev_units_del_rate) * dt$
 INIT $Deleted_Dev_Units = 0$
 INFLOWS:
 \Rightarrow $dev_units_del_rate = raw_dev_units_del_due_to_RC + dev_units_del_due_to_int$
 $Deleted_QAed_Deved_Units(t) = Deleted_QAed_Deved_Units(t - dt) + (QAed_dev_units_del_rate) * dt$
 INIT $Deleted_QAed_Deved_Units = 0$
 INFLOWS:
 \Rightarrow $QAed_dev_units_del_rate = QAed_units_del_due_to_RC + QAed_units_del_due_to_int$
 $QAed_Units_Deved_To_Test(t) = QAed_Units_Deved_To_Test(t - dt) + (QAed_deved_units_to_test) * dt$
 INIT $QAed_Units_Deved_To_Test = 0$
 INFLOWS:
 \Rightarrow $QAed_deved_units_to_test = dev_QA_rate$
 $Units_Developed(t) = Units_Developed(t - dt) + (dev_rate - dev_QA_rate - deved_units_deletion) * dt$
 INIT $Units_Developed = 0$
 INFLOWS:
 \Rightarrow $dev_rate = daily_MP_to_dev * dev_prod_ratio * dev_prod_rate * degree_of_concurrency * DT$
 DOCUMENT:
 Development rate (development units worked per day) is determined by three parameters: daily manpower allocated to development, development production ratio, and sequential constraint Sequential constraint (defined as degree of concurrency)
- OUTFLOWS:
 \Rightarrow $dev_QA_rate = (Units_Developed/dev_QA_duration) * (daily_MP_to_dev_QA / (daily_MP_to_dev_QA + 0.00001))$
 DOCUMENT:
 Number of development units that are quality assured per day
 \Rightarrow $deved_units_deletion = deved_units_del_due_to_RC + deved_units_del_due_to_int$
 $Units_QAed(t) = Units_QAed(t - dt) + (dev_QA_rate - QAed_deved_units_to_test - QAed_units_deletion) * dt$
 INIT $Units_QAed = 0$
 INFLOWS:

$$\text{dev_QA_rate} = (\text{Units_Developed}/\text{dev_QA_duration}) * (\text{daily_MP_to_dev_QA}/(\text{daily_MP_to_dev_QA}+0.00001))$$

DOCUMENT:

Number of development units that are quality assured per day

OUTFLOWS:

$$\text{QAed_deved_units_to_test} = \text{dev_QA_rate}$$

$$\text{QAed_units_deletion} = \text{QAed_units_del_due_to_RC} + \text{QAed_units_del_due_to_int}$$

$$\square \text{Units_To_Be_Developed}(t) = \text{Units_To_Be_Developed}(t - dt) + (\text{units_TBD_incoming_rate} - \text{dev_rate} - \text{dev_units_deletion}) * dt$$

INIT Units_To_Be_Developed = 0

INFLOWS:

$$\text{units_TBD_incoming_rate} = \text{QAed_spec_to_dev_rate} * \text{dev_units_per_reqs} + \text{dev_units_inc_due_to_int}$$

OUTFLOWS:

$$\text{dev_rate} = \text{daily_MP_to_dev} * \text{dev_prod_ratio} * \text{dev_prod_rate} * \text{degree_of_concurrency} * DT$$

DOCUMENT:

Development rate (development units worked per day) is determined by three parameters: daily manpower allocated to development, development production ratio, and sequential constraint Sequential constraint (defined as degree of concurrency)

$$\text{dev_units_deletion} = \text{raw_dev_units_del_due_to_RC} + \text{dev_units_del_due_to_int}$$

$$\circ \text{dev_prod_rate} = \text{actual_staff_prod_rate} / \text{LOC_per_dev_unit}$$

DOCUMENT:

Development units worked per day

$$\circ \text{dev_prod_ratio} = 1$$

$$\circ \text{dev_QA_duration} = 10$$

DOCUMENT:

Set to 10 working days [7]

$$\circ \text{dev_units_per_reqs} = \text{LOC_per_reqs}/\text{LOC_per_dev_unit}$$

$$\circ \text{frac_dev_pcvd_completed} = \text{Cum_Units_QAed} / \text{pcvd_total_dev_units}$$

$$\circ \text{LOC_per_dev_unit} = 60$$

DOCUMENT:

A development unit is set to 60 lines of source code [7]

$$\circ \text{pcvd_total_dev_units} = (\text{Pcvd_Project_Size} * 1000) / \text{LOC_per_dev_unit}$$

$$\circ \text{degree_of_concurrency} = \text{GRAPH}(\text{frac_dev_pcvd_completed})$$

(0.00, 1.00), (0.1, 1.00), (0.2, 1.00), (0.3, 0.7), (0.4, 0.7), (0.5, 0.7), (0.6, 0.7), (0.7, 0.5), (0.8, 0.5), (0.9, 0.5), (1, 0.5)

DOCUMENT:

Degree of concurrency (defined as the ratio of the number of development units ready for assignment and the number of development units that staff members are able to perform (i.e., sequential constrain

The value is determined project managers to simulate different degrees of sequential constraints

Fraction Project Completed

- $\text{frac_project_pcvd_completed} = \text{frac_spec_pcvd_completed} * \text{PC_weight_to_reqs} + \text{frac_dev_pcvd_completed} * \text{PC_weight_to_dev} + \text{frac_units_tested} * \text{PC_weight_to_SIT}$
- $\text{PC_weight_to_dev} = 0.75$
- $\text{PC_weight_to_reqs} = 0.25$
- $\text{PC_weight_to_SIT} = 0$
- $\text{project_complete} = \text{IF}(\text{frac_units_tested} * 100 > 98 \text{ AND } \text{defects_removed} = 1) \text{ THEN PAUSE ELSE } 0$

InterTEAM Interactions

- $\text{Detected_Dev_Ints}(t) = \text{Detected_Dev_Ints}(t - dt) + (\text{dev_int_det} - \text{dev_int_resolution}) * dt$
INIT $\text{Detected_Dev_Ints} = 0$
INFLOWS:
 - $\text{dev_int_det} = \text{IF}(\text{units_integration_rate} > 0.1) \text{ THEN } \text{ints_density} * \text{units_integration_rate} \text{ ELSE } \text{daily_MP_to_int_detection} / (\text{effort_to_detect_a_dev_int} * 1)$
 OUTFLOWS:
 - $\text{dev_int_resolution} = \text{Detected_Dev_Ints} / \text{int_resolution_delay}$
- $\text{Detected_Reqs_Ints}(t) = \text{Detected_Reqs_Ints}(t - dt) + (\text{reqs_int_det} - \text{reqs_int_resolution}) * dt$
INIT $\text{Detected_Reqs_Ints} = 0$
INFLOWS:
 - $\text{reqs_int_det} = \text{daily_MP_to_int_detection} / (\text{effort_to_detect_a_reqs_int} * 1)$
 OUTFLOWS:
 - $\text{reqs_int_resolution} = \text{Detected_Reqs_Ints} / \text{int_resolution_delay}$
- $\text{Dev_Ints_Resolved}(t) = \text{Dev_Ints_Resolved}(t - dt) + (\text{dev_int_resolution}) * dt$
INIT $\text{Dev_Ints_Resolved} = 0$
INFLOWS:
 - $\text{dev_int_resolution} = \text{Detected_Dev_Ints} / \text{int_resolution_delay}$
- $\text{Escaped_Reqs_Ints}(t) = \text{Escaped_Reqs_Ints}(t - dt) + (\text{reqs_ints_to_dev}) * dt$
INIT $\text{Escaped_Reqs_Ints} = 0$
INFLOWS:
 - $\text{reqs_ints_to_dev} = (\text{dev_rate} / \text{dev_units_per_reqs}) * \text{reqs_int_density}$
- $\text{Int_Detection_Effort}(t) = \text{Int_Detection_Effort}(t - dt) + (\text{int_det_effort_cum_rate}) * dt$
INIT $\text{Int_Detection_Effort} = 0$
INFLOWS:
 - $\text{int_det_effort_cum_rate} = \text{daily_MP_to_int_detection}$
- $\text{Reqs_Ints_Resolved}(t) = \text{Reqs_Ints_Resolved}(t - dt) + (\text{reqs_int_resolution}) * dt$
INIT $\text{Reqs_Ints_Resolved} = 0$
INFLOWS:
 - $\text{reqs_int_resolution} = \text{Detected_Reqs_Ints} / \text{int_resolution_delay}$
- $\text{Undetected_Dev_Ints}(t) = \text{Undetected_Dev_Ints}(t - dt) + (\text{dev_int_regen} + \text{ints_from_reqs} - \text{dev_int_det}) * dt$
INIT $\text{Undetected_Dev_Ints} = 0$
INFLOWS:

- $\text{dev_int_regen} = \text{mult_to_across_team_int_amp} * \text{across_team_interference_amplification} * (\text{frac_dev_int} * \text{dev_rate} * \text{dev_phase_interference_amplification} * \text{DELAY}(\text{dev_int_density}, 40))$
 $\text{ints_from_reqs} = \text{mult_to_across_team_int_amp} * \text{across_team_interference_amplification} * ((1 - \text{reqs_int_density}) * \text{dev_rate} * \text{frac_dev_int} + \text{reqs_int_density} * \text{dev_rate})$
- OUTFLOWS:
- $\text{dev_int_det} = \text{IF}(\text{units_integration_rate} > 0.1)$
 THEN $\text{ints_density} * \text{units_integration_rate}$
 ELSE $\text{daily_MP_to_int_detection} / (\text{effort_to_detect_a_dev_int} * 1)$
- $\text{Undetected_Reqs_Ints}(t) = \text{Undetected_Reqs_Ints}(t - dt) + (\text{reqs_int_gen} - \text{reqs_int_det} - \text{reqs_ints_to_dev}) * dt$
 INIT $\text{Undetected_Reqs_Ints} = 0$
- INFLOWS:
- $\text{reqs_int_gen} = \text{IF}(\text{frac_project_pcvd_completed} > 0.5)$
 THEN 0
 ELSE $\text{spec_rate} * \text{frac_reqs_int} * \text{across_team_interference_amplification} * \text{mult_to_across_team_int_amp}$
- OUTFLOWS:
- $\text{reqs_int_det} = \text{daily_MP_to_int_detection} / (\text{effort_to_detect_a_reqs_int} * 1)$
 $\text{reqs_ints_to_dev} = (\text{dev_rate} / \text{dev_units_per_reqs}) * \text{reqs_int_density}$
- $\text{daily_MP_to_int_detection} = \text{total_daily_MP} * \text{frac_daily_MP_to_int_detection}$
- $\text{dev_int_density} = \text{Undetected_Dev_Ints} / (\text{Cum_Units_Developed} * (1 - \text{frac_ints_detected}) + 0.00001)$
- $\text{frac_ints_detected} = \text{Detected_Dev_Ints} / (\text{Detected_Dev_Ints} + \text{Undetected_Dev_Ints} + 0.000001)$
- $\text{ints_density} = \text{Undetected_Dev_Ints} / (\text{Units_To_Be_Integrated} + 0.00001)$
- $\text{int_resolution_delay} = 5$
- DOCUMENT:
 Interteam interferences resolution delay
 Set to 5 working days based on Fujitsu's experience
- $\text{mult_to_across_team_int_amp} = 0.573$
- DOCUMENT:
 The values are set to model different percentages of rework incurred by multiple-team concurrent development
 More detailed explanations of F1, F2, and F3 are included in chapter 7
 F1: 0.213; F2: 0.427; F3: 0.573
- $\text{reqs_int_density} = \text{Undetected_Reqs_Ints} / (\text{Units_To_Be_Developed} / \text{dev_units_per_reqs} + 0.000001)$
- $\text{across_team_interference_amplification} = \text{GRAPH}(\text{number_of_teams})$
 (1.00, 0.00), (2.00, 1.00), (3.00, 1.08), (4.00, 1.20), (5.00, 1.38), (6.00, 1.53), (7.00, 1.73), (8.00, 1.98), (9.00, 2.25), (10.0, 2.69), (11.0, 3.30)
- DOCUMENT:
 Interteam interferences amplification caused by multiple-team concurrent development
 Modeled as a nonlinear function of the number of concurrent teams according to Fujitsu's experience

- ⊙ $\text{dev_phase_interference_amplification} = \text{GRAPH}(\text{frac_dev_pcvd_completed})$
 (0.00, 2.50), (0.1, 2.20), (0.2, 1.90), (0.3, 1.60), (0.4, 1.35), (0.5, 1.10), (0.6, 0.85), (0.7, 0.55), (0.8, 0.35), (0.9, 0.15), (1, 0.00)
 DOCUMENT:
 Interferences amplification along the dimension of development life cycle
 Based on Fujitsu's experience
- ⊙ $\text{effort_to_detect_a_dev_int} = \text{GRAPH}(\text{dev_int_density})$
 (0.00, 2.00), (0.1, 1.69), (0.2, 1.55), (0.3, 1.41), (0.4, 1.33), (0.5, 1.25), (0.6, 1.19), (0.7, 1.13), (0.8, 1.07), (0.9, 1.05), (1, 1.00)
 DOCUMENT:
 Average effort to detect an interteam development (design and coding) interference
 Modeled as a graph function of development interference density according to Fujitsu's experience
- ⊙ $\text{effort_to_detect_a_reqs_int} = \text{GRAPH}(\text{reqs_int_density})$
 (0.00, 0.203), (0.1, 0.165), (0.2, 0.14), (0.3, 0.13), (0.4, 0.12), (0.5, 0.115), (0.6, 0.11), (0.7, 0.108), (0.8, 0.104), (0.9, 0.102), (1, 0.1)
 DOCUMENT:
 Average effort to detect a requirements phase interteam interference
 Modeled as a graph function of requirements interference density according to Fujitsu's experience
- ⊙ $\text{frac_daily_MP_to_int_detection} = \text{GRAPH}(\text{IF}(\text{number_of_teams} \leq 1)$
 THEN 0
 ELSE $\text{frac_project_pcvd_completed}$)
 (0.00, 0.00), (0.1, 0.00), (0.2, 0.05), (0.3, 0.00), (0.4, 0.00), (0.5, 0.00), (0.6, 0.00), (0.7, 0.00), (0.8, 0.05), (0.9, 0.00), (1, 0.00)
 DOCUMENT:
 The fraction of daily manpower that is allocated to interteam interference detection
- ⊙ $\text{frac_dev_int} = \text{GRAPH}(\text{interteam_communication_overhead})$
 (0.00, 0.1), (0.1, 0.09), (0.2, 0.083), (0.3, 0.077), (0.4, 0.074), (0.5, 0.0705), (0.6, 0.068), (0.7, 0.066), (0.8, 0.064), (0.9, 0.062), (1, 0.06)
 DOCUMENT:
 The fraction of development tasks that are considered as interferences
 Modeled as a graph function of interteam communication overhead
 The general shape of the graph is based on Fujitsu's experience
- ⊙ $\text{frac_reqs_int} = \text{GRAPH}(\text{interteam_communication_overhead} * 100)$
 (0.00, 0.1), (1.00, 0.094), (2.00, 0.087), (3.00, 0.082), (4.00, 0.0765), (5.00, 0.072), (6.00, 0.069), (7.00, 0.066), (8.00, 0.064), (9.00, 0.062), (10.0, 0.06)
 DOCUMENT:
 The fraction of requirements specifications that are considered as interferences
 Modeled as a graph function of interteam communication overhead
 The general shape of the graph function is based on Fujitsu's experience

Overall Communication Overhead

- $\text{Cum_interteam_Comm_Overhead}(t) = \text{Cum_interteam_Comm_Overhead}(t - dt) +$
 $(\text{interteam_comm_cum_rate}) * dt$
 INIT $\text{Cum_interteam_Comm_Overhead} = 0$
 INFLOWS:

- $\text{interteam_comm_cum_rate} = \text{mult_to_interteam_comm_overhead} * \text{interteam_communication_overhead}$
- $\text{Cum_Intrateam_Comm_Overhead}(t) = \text{Cum_Intrateam_Comm_Overhead}(t - dt) + (\text{intrateam_comm_factor_cum_rate}) * dt$
 INIT Cum_Intrateam_Comm_Overhead = 0
 INFLOWS:
 $\text{intrateam_comm_factor_cum_rate} = \text{intrateam_communication_overhead} * \text{number_of_teams}$
- $\text{Cum_Overall_Comm_Overhead}(t) = \text{Cum_Overall_Comm_Overhead}(t - dt) + (\text{overall_comm_overhead_cum_rate}) * dt$
 INIT Cum_Overall_Comm_Overhead = 0
 INFLOWS:
 $\text{overall_comm_overhead_cum_rate} = \text{overall_communication_overhead}$
- $\text{Cum_Team_Size}(t) = \text{Cum_Team_Size}(t - dt) + (\text{team_size_cum_rate}) * dt$
 INIT Cum_Team_Size = 0
 INFLOWS:
 $\text{team_size_cum_rate} = \text{average_team_size}$
- $\text{average_interteam_comm_overhead} = \text{Cum_interteam_Comm_Overhead}/(\text{TIME}+0.00001)$
- $\text{average_intrateam_comm_overhead} = \text{Cum_Intrateam_Comm_Overhead}/(\text{TIME}+0.00001)$
- $\text{average_overall_comm_overhead} = \text{Cum_Overall_Comm_Overhead}/(\text{TIME}+0.00001)$
- $\text{average_team_size} = \text{current_WF} / \text{number_of_teams}$
 DOCUMENT:
 Total number of current work force level divided by the number of concurrent teams
- $\text{interteam_to_intrateam_comm_ratio} = (100 * \text{average_interteam_comm_overhead}) / (100 * \text{average_intrateam_comm_overhead} + 0.00001)$
- $\text{mult_to_interteam_comm_overhead} = 20$
 DOCUMENT:
 The value is set to model different interteam-to-intrateam communication overhead ratio
 More detailed explanations are included in chapter 7
 M1: 0.25; M2: 1.0; M3: 2.0
- $\text{number_of_teams} = 10$
 DOCUMENT:
 Total number of concurrent development teams
- $\text{overall_communication_overhead} = \text{MIN}(1, \text{intrateam_communication_overhead} + \text{interteam_communication_overhead} * \text{mult_to_interteam_comm_overhead})$
- $\text{project_average_team_size} = \text{Cum_Team_Size}/(\text{TIME}+0.00001)$
- $\text{interteam_communication_overhead} = \text{GRAPH}(\text{number_of_teams})$
 (1.00, 0.00), (2.00, 0.004), (3.00, 0.0085), (4.00, 0.014), (5.00, 0.021), (6.00, 0.028), (7.00, 0.0355), (8.00, 0.043), (9.00, 0.0505), (10.0, 0.063)
 DOCUMENT:
 Interteam communication overhead is modeled as a function of the number of concurrent teams
- $\text{intrateam_communication_overhead} = \text{GRAPH}(\text{average_team_size})$
 (0.00, 0.00), (5.00, 0.015), (10.0, 0.06), (15.0, 0.135), (20.0, 0.24), (25.0, 0.375), (30.0, 0.54)
 DOCUMENT:
 Intrateam communication overhead is modeled as a function of average team size

Planning

- $\text{average_component_size} = (\text{initial_num_of_reqs} * \text{LOC_per_reqs}) / \text{number_of_components}$
- $\text{average_WF} = \text{ROUND}(\text{initial_effort_estimate}/\text{initial_duration_estimate})$
- $\text{BRAK_factor} = 0$
DOCUMENT:
Breakage percentage: a COCOMO 2.0 Requirements Volatility measure
- $\text{estimate_of_project_size} = 128$
DOCUMENT:
Project size in KLOC
- $\text{initial_duration_estimate} = 19 * 33.3$
DOCUMENT:
Initial estimate of project duration
One month is considered as equal to 19 working days
- $\text{initial_effort_estimate} = 19 * (46.1 + 658.9)$
DOCUMENT:
Initial estimate of project effort
Derived from a nominal 128 KLOC COCOMO 2.0 project; 46.1 person-months for requirements; 658.9 person-months for development and integration and test
One person-month is considered as equal to 19 person-days
- $\text{initial_exp_WF} = (\text{average_WF} * \text{init_staffing_factor}) * (\text{init_pct_staff_exp}/100)$
- $\text{initial_new_WF} = (\text{average_WF} * \text{init_staffing_factor}) * (1 - \text{init_pct_staff_exp}/100)$
- $\text{initial_num_of_reqs} = (\text{estimate_of_project_size} * 1000) / \text{LOC_per_reqs}$
- $\text{init_pct_staff_exp} = 100$
- $\text{init_planned_effort_to_dev_phase} = (\text{initial_effort_estimate} * \text{pct_effort_to_dev}) / 100$
- $\text{init_planned_effort_to_reqs} = \text{initial_effort_estimate} * (\text{pct_effort_to_reqs}/100)$
- $\text{init_planned_effort_to_SIT} = \text{initial_effort_estimate} * (\text{pct_effort_to_SIT}/100)$
- $\text{init_staffing_factor} = 0.37$
- $\text{LOC_per_reqs} = 125$
DOCUMENT:
A requirements unit is assumed to be 125 LOC large
- $\text{number_of_components} = 128$
- $\text{pct_effort_to_dev} = 67.3$
DOCUMENT:
The percentage of project effort that is allocated to the development (design and coding) phase
Based on COCOMO 2.0 [23]
- $\text{pct_effort_to_reqs} = 6.5$
DOCUMENT:
The percentage of project effort that is allocated to the requirements phase
Based on COCOMO 2.0 [23]
- $\text{pct_effort_to_SIT} = 26.2$
DOCUMENT:
The percentage of project effort that is allocated to system integration and test
Based on COCOMO 2.0 [23]
- $\text{Unplanned_Reqs_Change} = \text{initial_num_of_reqs} * (\text{BRAK_factor}/100)$

Project Control

- $\text{Planned_Project_Duration}(t) = \text{Planned_Project_Duration}(t - dt) + (\text{project_duration_change_rate}) * dt$
 INIT $\text{Planned_Project_Duration} = \text{initial_duration_estimate}$

INFLOWS:

- $\text{project_duration_change_rate} = \text{IF}(\text{frac_project_pcvd_completed} < 1 \text{ AND } \text{Planned_Project_Effort} > \text{INIT}(\text{Planned_Project_Effort}))$
 THEN $(\text{target_project_duration} - \text{Planned_Project_Duration}) / (\text{sched_adjustment_time} / \text{DT})$
 ELSE 0

- $\text{Planned_Project_Effort}(t) = \text{Planned_Project_Effort}(t - dt) + (\text{PPE_change_rate}) * dt$
 INIT $\text{Planned_Project_Effort} = \text{initial_effort_estimate}$

INFLOWS:

- $\text{PPE_change_rate} = (\text{target_project_effort} - \text{Planned_Project_Effort}) /$
 $(\text{planned_project_effort_adj_time} / \text{DT})$

- $\text{Project_Effort_Expenditure}(t) = \text{Project_Effort_Expenditure}(t - dt) + (\text{project_effort_cum_rate}) * dt$
 INIT $\text{Project_Effort_Expenditure} = 0$

INFLOWS:

- $\text{project_effort_cum_rate} = \text{IF}(\text{frac_project_pcvd_completed} > 0.95 \text{ AND } \text{defects_removed} = 1)$
 THEN 0
 ELSE $\text{current_WF} * \text{DT}$

- $\text{Project_Elapsed_Time}(t) = \text{Project_Elapsed_Time}(t - dt) + (\text{PET_inc_rate}) * dt$
 INIT $\text{Project_Elapsed_Time} = 0$

INFLOWS:

- $\text{PET_inc_rate} = \text{IF}(\text{project_complete})$
 THEN 0
 ELSE DT

- $\text{MP_excess_absorbed} = \text{MAX}(0,$
 $\text{frac_MP_excess_absorbed} * (\text{Planned_Project_Effort} - \text{Project_Effort_Expenditure}) -$
 $\text{pcvd_project_effort_needed})$

- $\text{MP_gap_handled} = \text{IF}(\text{pcvd_project_effort_gap} > 0)$
 THEN $\text{MIN}(\text{pcvd_project_effort_gap}, \text{max_MP_shortage_to_be_handled})$
 ELSE 0

- $\text{pcvd_project_effort_gap} = \text{pcvd_project_effort_needed} -$
 $(\text{Planned_Project_Effort} - \text{Project_Effort_Expenditure})$

- $\text{pcvd_project_effort_needed} =$
 $\text{pcvd_reqs_phase_effort_needed} + \text{pcvd_dev_phase_effort_needed} + \text{pcvd_SIT_effort_needed}$

- $\text{planned_project_effort_adj_time} = 3$

DOCUMENT:

The delay in adjusting the perceived project effort
 Set to 3 working days [7]

- $\text{project_effort_gap_reported} = \text{pcvd_project_effort_gap} - \text{MP_gap_handled} + \text{MP_excess_absorbed}$

- $\text{project_time_remaining} = \text{MAX}(\text{Planned_Project_Duration} - \text{Project_Elapsed_Time}, 0)$

- $\text{remaining_project_effort} = \text{MAX}(0, \text{target_project_effort} - \text{Project_Effort_Expenditure})$

- $\text{schedule_pressure} = \text{SMTH1}(\text{pcvd_project_effort_gap} / 100, 40)$

- sched_adjustment_time = 3
DOCUMENT:
The delay in adjusting the planned project schedule
Set to 3 working days (i.e., the same as project effort adjustment delay)
- target_project_duration = Project_Elapsed_Time+time_needed
- target_project_effort = Planned_Project_Effort + project_effort_gap_reported
- target_WF = IF(project_time_remaining>10)
THEN (Planned_Project_Effort-Project_Effort_Expenditure+project_effort_gap_reported) /
project_time_remaining
ELSE (Planned_Project_Effort-Project_Effort_Expenditure+project_effort_gap_reported)/10
- time_needed = IF(current_WF+desired_new_staff>average_WF)
THEN remaining_project_effort / (current_WF+desired_new_staff)
ELSE remaining_project_effort / average_WF
- frac_MP_excess_absorbed =
GRAPH(pcvd_project_effort_needed/(Planned_Project_Effort-Project_Effort_Expenditure+0.00001))
(0.00, 0.00), (0.1, 0.2), (0.2, 0.4), (0.3, 0.55), (0.4, 0.7), (0.5, 0.8), (0.6, 0.9), (0.7, 0.95), (0.8, 1.00), (0.9,
1.00), (1, 1.00)
- weight_to_actual_project_effort_needed = GRAPH(frac_project_pcvd_completed)
(0.00, 0.00), (0.1, 0.1), (0.2, 0.2), (0.3, 0.3), (0.4, 0.4), (0.5, 0.5), (0.6, 0.6), (0.7, 0.7), (0.8, 0.8), (0.9,
0.9), (1, 1.00)

Project Effort 1

- Cum_Dev_Defects_Correction_Effort(t) = Cum_Dev_Defects_Correction_Effort(t - dt) +
(dev_defects_correction_effort_cum_rate) * dt
INIT Cum_Dev_Defects_Correction_Effort = 0
INFLOWS:
 dev_defects_correction_effort_cum_rate = (DT*daily_MP_to_dev_defect_correction) /
(daily_MP_factor+0.000001)
- Cum_Dev_Effort(t) = Cum_Dev_Effort(t - dt) + (dev_MP_expending_rate) * dt
INIT Cum_Dev_Effort = 0
INFLOWS:
 dev_MP_expending_rate = (DT*daily_MP_to_dev) / (daily_MP_factor+0.000001)
- Cum_Dev_QA_Effort(t) = Cum_Dev_QA_Effort(t - dt) + (dev_QA_MP_expending_rate) * dt
INIT Cum_Dev_QA_Effort = 0
INFLOWS:
 dev_QA_MP_expending_rate = (DT*daily_MP_to_dev_QA) / (daily_MP_factor+0.00001)
- Reqs_Defects_Correction_Effort(t) = Reqs_Defects_Correction_Effort(t - dt) +
(reqs_defects_correction_effort_cum_rate) * dt
INIT Reqs_Defects_Correction_Effort = 0
INFLOWS:
 reqs_defects_correction_effort_cum_rate = daily_MP_to_spec_defect_correction /
(daily_MP_factor+0.000001)
- Reqs_Spec_Effort(t) = Reqs_Spec_Effort(t - dt) + (spec_effort_cum_rate) * dt
INIT Reqs_Spec_Effort = 0
INFLOWS:

- ☞ $\text{spec_effort_cum_rate} = \text{daily_MP_to_spec} / (\text{daily_MP_factor} + 0.0000001)$
- $\text{Spec_QA_Effort}(t) = \text{Spec_QA_Effort}(t - dt) + (\text{spec_QA_effort_cum_rate}) * dt$
 INIT $\text{Spec_QA_Effort} = 0$
 INFLOWS:
- ☞ $\text{spec_QA_effort_cum_rate} = \text{daily_MP_to_spec_QA} / (\text{daily_MP_factor} + 0.0000001)$
- $\text{Training_Effort}(t) = \text{Training_Effort}(t - dt) + (\text{training_effort_increase_rate}) * dt$
 INIT $\text{Training_Effort} = 0$
 INFLOWS:
- ☞ $\text{training_effort_increase_rate} = \text{current_WF} * (DT * \text{training_time})$

Project Effort 2

- $\text{Cum_Dev_Phase_Effort}(t) = \text{Cum_Dev_Phase_Effort}(t - dt) + (\text{daily_dev_phase_MP_exp_rate}) * dt$
 INIT $\text{Cum_Dev_Phase_Effort} = 0$
 INFLOWS:
- ☞ $\text{daily_dev_phase_MP_exp_rate} = (DT * \text{daily_MP_to_dev_phase}) / (\text{daily_MP_factor} + 0.000001)$
- $\text{Cum_Reqs_Phase_Effort}(t) = \text{Cum_Reqs_Phase_Effort}(t - dt) + (\text{reqs_effort_expending_rate}) * dt$
 INIT $\text{Cum_Reqs_Phase_Effort} = 0$
 INFLOWS:
- ☞ $\text{reqs_effort_expending_rate} = \text{daily_MP_to_reqs_phase} / (\text{daily_MP_factor} + 0.000001)$
- $\text{Cum_SIT_Effort}(t) = \text{Cum_SIT_Effort}(t - dt) + (\text{daily_SIT_MP_expending_rate}) * dt$
 INIT $\text{Cum_SIT_Effort} = 0$
 INFLOWS:
- ☞ $\text{daily_SIT_MP_expending_rate} = (DT * \text{daily_MP_to_SIT_phase}) / (\text{daily_MP_factor} + 0.00001)$
- $\text{Defects_FIT_Correction_Effort}(t) = \text{Defects_FIT_Correction_Effort}(t - dt) + (\text{defects_FIT_correction_effort_cum_rate}) * dt$
 INIT $\text{Defects_FIT_Correction_Effort} = 0$
 INFLOWS:
- ☞ $\text{defects_FIT_correction_effort_cum_rate} = (DT * \text{daily_MP_to_defects_FIT_correction}) / (\text{daily_MP_factor} + 0.00001)$
- $\text{System_Integration_Effort}(t) = \text{System_Integration_Effort}(t - dt) + (\text{SI_effort_cum_rate}) * dt$
 INIT $\text{System_Integration_Effort} = 0$
 INFLOWS:
- ☞ $\text{SI_effort_cum_rate} = (DT * \text{daily_MP_to_integration}) / (\text{daily_MP_factor} + 0.00001)$
- $\text{System_Test_Effort}(t) = \text{System_Test_Effort}(t - dt) + (\text{system_test_MP_expending_rate}) * dt$
 INIT $\text{System_Test_Effort} = 0$
 INFLOWS:
- ☞ $\text{system_test_MP_expending_rate} = (DT * \text{daily_MP_to_test}) / (\text{daily_MP_factor} + 0.00001)$
- $\text{cumulative_project_effort} = \text{Int_Detection_Effort} + \text{Change_Rework_Overhead} + \text{Cum_Reqs_Phase_Effort} + \text{Cum_Dev_Phase_Effort} + \text{Cum_SIT_Effort}$

Project Scope Change

- $\text{Change_Rework_Overhead}(t) = \text{Change_Rework_Overhead}(t - dt) + (\text{daily_MP_to_reqs_change_rework}) * dt$
 INIT $\text{Change_Rework_Overhead} = 0$

INFLOWS:

$$\text{daily_MP_to_reqs_change_rework} = (\text{reqs_change_rate} * \text{rework_cost_ratio} * \text{nominal_rework_cost}) / (\text{daily_MP_factor} + 0.000001)$$

$$\square \text{Cum_Discovered_Reqs}(t) = \text{Cum_Discovered_Reqs}(t - dt) + (\text{reqs_discovery}) * dt$$

INIT Cum_Discovered_Reqs = 0

INFLOWS:

$$\text{reqs_discovery} = \text{unplanned_reqs_discovery}$$

$$\square \text{Cum_Reqs_Change}(t) = \text{Cum_Reqs_Change}(t - dt) + (\text{reqs_change_rate}) * dt$$

INIT Cum_Reqs_Change = 0

INFLOWS:

$$\text{reqs_change_rate} = \text{Discovered_Reqs} / (\text{unplanned_reqs_inc_delay} / DT)$$

$$\square \text{Discovered_Reqs}(t) = \text{Discovered_Reqs}(t - dt) + (\text{unplanned_reqs_discovery} - \text{reqs_change_rate}) * dt$$

INIT Discovered_Reqs = 0

INFLOWS:

$$\text{unplanned_reqs_discovery} = \text{Unplanned_Reqs} * (DT * \text{frac_unplanned_reqs_discovered_per_day_C1}) / 100$$

OUTFLOWS:

$$\text{reqs_change_rate} = \text{Discovered_Reqs} / (\text{unplanned_reqs_inc_delay} / DT)$$

$$\square \text{Pcvd_Project_Size}(t) = \text{Pcvd_Project_Size}(t - dt) + (\text{PPS_inc} - \text{PPS_dec}) * dt$$

INIT Pcvd_Project_Size = estimate_of_project_size

INFLOWS:

$$\text{PPS_inc} = (\text{LOC_per_reqs} / 1000) * (\text{reqs_change_rate} * \text{frac_reqs_addition} + \text{raw_reqs_inc_due_to_reqs_int})$$

OUTFLOWS:

$$\text{PPS_dec} = (\text{LOC_per_reqs} / 1000) * (\text{reqs_change_rate} * (1 - \text{frac_reqs_addition}) + \text{reqs_deletion_due_to_int})$$

$$\square \text{Unplanned_Reqs}(t) = \text{Unplanned_Reqs}(t - dt) + (-\text{unplanned_reqs_discovery}) * dt$$

INIT Unplanned_Reqs = Unplanned_Reqs_Change

OUTFLOWS:

$$\text{unplanned_reqs_discovery} = \text{Unplanned_Reqs} * (DT * \text{frac_unplanned_reqs_discovered_per_day_C1}) / 100$$

frac_reqs_addition = 1

nominal_rework_cost = 4

pct_unplanned_reqs_discovered = 100 * Cum_Discovered_Reqs / (INIT(Unplanned_Reqs) + 0.0001)

project_scope_change_percentage = 100 * (Pcvd_Project_Size - INIT(Pcvd_Project_Size)) / INIT(Pcvd_Project_Size)

unplanned_reqs_inc_delay = 10

DOCUMENT:

The delay in incorporating unplanned requirements into the project
Set to 10 working days

- $\text{frac_unplanned_reqs_discovered_per_day_C1} = \text{GRAPH}(\text{frac_project_pcvd_completed})$
 (0.00, 1.00), (0.1, 1.00), (0.2, 1.00), (0.3, 1.00), (0.4, 1.00), (0.5, 1.00), (0.6, 1.00), (0.7, 1.00), (0.8, 1.00), (0.9, 1.00), (1, 1.00)
DOCUMENT:
 The fraction of unplanned requirements discovery pattern (pattern C1)
 More detailed explanations are included in chapter 7
- $\text{frac_unplanned_reqs_discovered_per_day_C2} = \text{GRAPH}(\text{frac_project_pcvd_completed})$
 (0.00, 0.00), (0.1, 0.1), (0.2, 0.5), (0.3, 0.95), (0.4, 1.50), (0.5, 1.50), (0.6, 1.50), (0.7, 1.50), (0.8, 1.50), (0.9, 1.50), (1, 1.50)
- $\text{frac_unplanned_reqs_discovered_per_day_C3} = \text{GRAPH}(\text{frac_project_pcvd_completed})$
 (0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.1), (0.4, 0.3), (0.5, 0.7), (0.6, 1.20), (0.7, 2.00), (0.8, 2.00), (0.9, 2.00), (1, 2.00)
- $\text{rework_cost_ratio} = \text{GRAPH}(\text{frac_project_pcvd_completed})$
 (0.00, 1.00), (0.1, 5.00), (0.2, 5.00), (0.3, 5.00), (0.4, 10.0), (0.5, 10.0), (0.6, 10.0), (0.7, 15.0), (0.8, 20.0), (0.9, 20.0), (1, 20.0)
DOCUMENT:
 Overhead to incorporate requirements change during the requirements phase: during the design stage
 during the coding stage: during the test stage = 1:5:10:20

Project Scope Change Due To Requirements Change

- $\text{deved_units_del_due_to_int} = \text{dev_deletion_due_to_int} * \text{frac_deved_units}$
- $\text{deved_units_del_due_to_RC} = \text{dev_units_deletion_due_to_RC} * \text{frac_deved_units}$
- $\text{dev_change_due_to_int} = \text{dev_int_resolution}$
DOCUMENT:
 Development units change due to interference resolution; the resolution of requirements interferences and the resolution of development interferences
- $\text{dev_deletion_due_to_int} = \text{dev_change_due_to_int} * (1 - \text{frac_dev_units_addition})$
- $\text{dev_units_deletion_due_to_RC} = \text{reqs_deletion_due_to_RC} * \text{dev_units_per_reqs}$
- $\text{dev_units_del_due_to_int} = \text{dev_deletion_due_to_int} * \text{frac_raw_dev_units}$
- $\text{dev_units_inc_due_to_int} = \text{dev_change_due_to_int} * \text{frac_dev_units_addition}$
- $\text{frac_deved_units} = \text{Units_Developed} / (\text{Units_To_Be_Developed} + \text{Units_Developed} + \text{Units_QAed} + 0.000001)$
- $\text{frac_dev_int_from_reqs_int} = 0.5$
- $\text{frac_dev_units_addition} = 1$
- $\text{frac_QAed_spec} = \text{IF}(\text{Raw_Reqs} + \text{Reqs_Spec} + \text{QAed_Reqs_Spec} = 0)$
 THEN 0
 ELSE $\text{QAed_Reqs_Spec} / (\text{Raw_Reqs} + \text{Reqs_Spec} + \text{QAed_Reqs_Spec})$
- $\text{frac_QAed_units} = \text{Units_QAed} / (\text{Units_To_Be_Developed} + \text{Units_Developed} + \text{Units_QAed} + 0.00001)$
- $\text{frac_raw_dev_units} = \text{Units_To_Be_Developed} / (\text{Units_To_Be_Developed} + \text{Units_Developed} + \text{Units_QAed} + 0.00001)$
- $\text{frac_raw_reqs} = \text{IF}(\text{Raw_Reqs} + \text{Reqs_Spec} + \text{QAed_Reqs_Spec} = 0)$
 THEN 0
 ELSE $\text{Raw_Reqs} / (\text{Raw_Reqs} + \text{Reqs_Spec} + \text{QAed_Reqs_Spec})$

- ☞ $\text{spec_defect_escape_rate} = \text{spec_defects_detection_rate} * (1 - \text{spec_QA_effectiveness}) / (\text{spec_QA_effectiveness} + 0.00001)$
- $\text{Spec_Defects_Bad_Fixes}(t) = \text{Spec_Defects_Bad_Fixes}(t - dt) + (\text{spec_defects_bad_fixes_rate}) * dt$
INIT $\text{Spec_Defects_Bad_Fixes} = 0$
INFLOWS:
☞ $\text{spec_defects_bad_fixes_rate} = \text{spec_defect_fixing_rate} * \text{spec_defects_bad_fixes_ratio}$
- $\text{Spec_Defects_Fixed}(t) = \text{Spec_Defects_Fixed}(t - dt) + (\text{spec_defect_fixing_rate}) * dt$
INIT $\text{Spec_Defects_Fixed} = 0$
INFLOWS:
☞ $\text{spec_defect_fixing_rate} = (1 - \text{spec_defects_bad_fixes_ratio}) * (\text{daily_MP_to_spec_defect_correction} * DT) / \text{MP_needed_to_fix_a_spec_defect}$
- $\text{post_QA_spec_defect_density} = (\text{Spec_Defects_Bad_Fixes} + \text{Escaped_Spec_Defects}) / (\text{Cum_QAed_Reqs_Spec} + 0.00001)$
- $\text{pre_QA_spec_defect_density} = \text{Spec_Defects} / (\text{Reqs_Spec} + 0.00001)$
- $\text{reqs_defects_per_KLOC} = 5$
DOCUMENT: Requirements defects per KLOC = 5/KDSI [Boehm 81]
- $\text{spec_defects_bad_fixes_ratio} = 0.12$
- $\text{spec_QA_effectiveness} = \text{GRAPH}(\text{daily_MP_to_spec_QA})$
(0.00, 0.00), (0.1, 0.155), (0.2, 0.32), (0.3, 0.49), (0.4, 0.625), (0.5, 0.725), (0.6, 0.82), (0.7, 0.87), (0.8, 0.895), (0.9, 0.9), (1, 0.9)

Requirements Manpower Allocation

- $\text{average_daily_MP_per_staff} = 1$
- $\text{daily_MP_factor} = \text{average_daily_MP_per_staff} * \text{average_productive_time}$
- $\text{daily_MP_to_reqs_phase} = \text{IF}(\text{Spec_Defects} < 0.01 \text{ AND } \text{Detected_Spec_Defects} < 0.01 \text{ AND } \text{frac_spec_pcvd_completed} > 0.99)$
THEN 0
ELSE $\text{frac_daily_MP_to_reqs} * \text{net_total_daily_MP}$
- $\text{daily_MP_to_spec} = \text{MAX}(0, \text{daily_MP_to_reqs_phase} - \text{daily_MP_to_spec_QA} - \text{daily_MP_to_spec_defect_correction})$
- $\text{daily_MP_to_spec_defect_correction} = \text{MP_needed_to_fix_a_spec_defect} * \text{desired_spec_defect_correction_rate}$
- $\text{daily_MP_to_spec_QA} = \text{daily_MP_to_reqs_phase} * \text{Actual_Frac_MP_On_QA}$
- $\text{desired_spec_defect_correction_rate} = \text{Detected_Spec_Defects} / \text{spec_defect_correction_delay}$
- $\text{MP_needed_to_fix_a_spec_defect} = 0.5/8$
DOCUMENT:
Manpower needed to fix a specification defect
Set to 0.5 staff hours [50]
1 day = 8 hours
- $\text{net_total_daily_MP} = \text{total_daily_MP} - \text{daily_MP_to_reqs_change_rework} - \text{daily_MP_to_int_detection}$
- $\text{Reqs_Phase_Complete} = \text{IF}(\text{frac_spec_pcvd_completed} > 0.99)$
THEN 1
ELSE 0

- $\text{spec_and_QA_complete} = \text{IF}(\text{Raw_Reqs} < 0.1 \text{ AND } \text{Reqs_Spec} < 0.1)$
THEN 1
ELSE 0
- $\text{spec_defect_correction_delay} = 5$
- $\text{total_daily_MP} = \text{current_WF} * \text{daily_MP_factor}$
- $\text{frac_daily_MP_to_reqs} = \text{GRAPH}(\text{frac_spec_pcvd_completed})$
(0.00, 1.00), (0.1, 1.00), (0.2, 1.00), (0.3, 1.00), (0.4, 0.996), (0.5, 0.978), (0.6, 0.942), (0.7, 0.852),
(0.8, 0.726), (0.9, 0.456), (1, 0.00)
DOCUMENT:
The fraction of daily manpower that is allocated to the requirements phase

Requirements Manpower Needed

- $\text{Cum_Spec}(t) = \text{Cum_Spec}(t - dt) + (\text{spec_cum_rate}) * dt$
INIT Cum_Spec = 0
INFLOWS:
 $\text{spec_cum_rate} = \text{spec_rate}$
- $\text{actual_spec_MP_needed} = \text{reqs_remaining_to_be_specified} / (\text{actual_spec_productivity} + 0.00001)$
- $\text{actual_spec_productivity} = \text{IF}(\text{Reqs_Spec_Effort} > 0)$
THEN $\text{Cum_Spec} / (\text{Reqs_Spec_Effort} + 0.000001)$
ELSE $\text{planned_spec_productivity}$
- $\text{current_planned_reqs_phase_effort} =$
 $\text{init_planned_effort_to_reqs} * (\text{pcvd_total_dev_units} / \text{INIT}(\text{pcvd_total_dev_units}))$
- $\text{pcvd_reqs_phase_effort_needed} =$
 $\text{weight_to_actual_reqs_effort_needed} * (\text{actual_spec_MP_needed} + \text{spec_defect_correction_effort_needed} + \text{spec_QA_MP_needed}) +$
 $(1 - \text{weight_to_actual_reqs_effort_needed}) * \text{reqs_phase_effort_remaining}$
- $\text{planned_spec_productivity} = \text{INIT}(\text{Raw_Reqs}) /$
 $(\text{init_planned_effort_to_reqs} * (1 - \text{Actual_Frac_MP_On_QA}))$
- $\text{reqs_phase_effort_remaining} = \text{MAX}(0, \text{current_planned_reqs_phase_effort} -$
 $\text{Cum_Reqs_Phase_Effort})$
- $\text{reqs_remaining_to_be_specified} = \text{MAX}(\text{Pcvd_Project_Size} * 1000 / \text{LOC_per_reqs} - \text{Cum_Spec}, 0)$
- $\text{spec_defect_correction_effort_needed} = \text{Detected_Spec_Defects} * \text{MP_needed_to_fix_a_spec_defect}$
- $\text{spec_QA_MP_needed} = (\text{actual_spec_MP_needed} / (1 - \text{Actual_Frac_MP_On_QA})) *$
 $\text{Actual_Frac_MP_On_QA}$
- $\text{weight_to_actual_reqs_effort_needed} = \text{GRAPH}(\text{frac_spec_pcvd_completed})$
(0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.00), (0.4, 0.00), (0.5, 0.00), (0.6, 0.2), (0.7, 0.4), (0.8, 0.6),
(0.9, 0.8), (1, 1.00)

Requirements Work Flow

- $\text{Cum_QAed_Reqs_Spec}(t) = \text{Cum_QAed_Reqs_Spec}(t - dt) + (\text{QAed_reqs_spec_cum_rate} -$
 $\text{QAed_spec_del_rate}) * dt$
INIT Cum_QAed_Reqs_Spec = 0
INFLOWS:
 $\text{QAed_reqs_spec_cum_rate} = \text{spec_QA_rate}$

OUTFLOWS:

$$\text{QAed_spec_del_rate} = \text{QAed_spec_del_due_to_RC} + \text{QAed_spec_del_due_to_int}$$

$$\square \text{Cum_Reqs_Spec}(t) = \text{Cum_Reqs_Spec}(t - dt) + (\text{reqs_spec_cum_rate} - \text{spec_del_rate}) * dt$$

INIT Cum_Reqs_Spec = 0

INFLOWS:

$$\text{reqs_spec_cum_rate} = \text{spec_rate}$$

OUTFLOWS:

$$\text{spec_del_rate} = \text{spec_del_due_to_RC} + \text{spec_del_due_to_int}$$

$$\square \text{Deleted_QAed_Spec}(t) = \text{Deleted_QAed_Spec}(t - dt) + (\text{QAed_spec_del_rate}) * dt$$

INIT Deleted_QAed_Spec = 0

INFLOWS:

$$\text{QAed_spec_del_rate} = \text{QAed_spec_del_due_to_RC} + \text{QAed_spec_del_due_to_int}$$

$$\square \text{Deleted_Raw_Reqs}(t) = \text{Deleted_Raw_Reqs}(t - dt) + (\text{raw_reqs_del_rate}) * dt$$

INIT Deleted_Raw_Reqs = 0

INFLOWS:

$$\text{raw_reqs_del_rate} = \text{raw_reqs_del_due_to_RC} + \text{raw_reqs_del_due_to_int}$$

$$\square \text{Deleted_Spec}(t) = \text{Deleted_Spec}(t - dt) + (\text{spec_del_rate}) * dt$$

INIT Deleted_Spec = 0

INFLOWS:

$$\text{spec_del_rate} = \text{spec_del_due_to_RC} + \text{spec_del_due_to_int}$$

$$\square \text{QAed_Reqs_Spec}(t) = \text{QAed_Reqs_Spec}(t - dt) + (\text{spec_QA_rate} - \text{QAed_spec_to_dev_rate} - \text{QAed_spec_deletion}) * dt$$

INIT QAed_Reqs_Spec = 0

INFLOWS:

$$\text{spec_QA_rate} = \text{Reqs_Spec} / (\text{average_QA_delay}/DT) * \text{daily_MP_to_spec_QA}/(\text{daily_MP_to_spec_QA} + 0.00001)$$

DOCUMENT:

Number of requirements reviewed per day

OUTFLOWS:

$$\text{QAed_spec_to_dev_rate} = \text{QAed_Reqs_Spec}/(\text{QAed_spec_to_dev_delay}/DT)$$

$$\text{QAed_spec_deletion} = \text{QAed_spec_del_due_to_RC} + \text{QAed_spec_del_due_to_int}$$

$$\square \text{QAed_Reqs_Spec_To_Dev_Phase}(t) = \text{QAed_Reqs_Spec_To_Dev_Phase}(t - dt) + (\text{QAed_spec_to_dev_rate}) * dt$$

INIT QAed_Reqs_Spec_To_Dev_Phase = 0

INFLOWS:

$$\text{QAed_spec_to_dev_rate} = \text{QAed_Reqs_Spec}/(\text{QAed_spec_to_dev_delay}/DT)$$

$$\square \text{Raw_Reqs}(t) = \text{Raw_Reqs}(t - dt) + (\text{reqs_incoming_rate} - \text{spec_rate} - \text{raw_reqs_deletion}) * dt$$

INIT Raw_Reqs = INIT(Pcvd_Project_Size)*1000 / LOC_per_reqs

INFLOWS:

$$\text{reqs_incoming_rate} = (\text{raw_reqs_inc_due_to_reqs_change} + \text{raw_reqs_inc_due_to_reqs_int})$$

OUTFLOWS:

- ☞ $\text{spec_rate} = \text{IF}(\text{Raw_Reqs} > 0)$
 THEN $\text{daily_MP_to_spec} * \text{spec_prod_rate} * \text{DT}$
 ELSE 0
 DOCUMENT:
 Requirements specification rate (i.e., requirements specified per day)
- ☞ $\text{raw_reqs_deletion} = \text{raw_reqs_del_due_to_RC} + \text{raw_reqs_del_due_to_int}$
- $\text{Reqs_Spec}(t) = \text{Reqs_Spec}(t - dt) + (\text{spec_rate} - \text{spec_QA_rate} - \text{spec_deletion}) * dt$
 INIT $\text{Reqs_Spec} = 0$
 INFLOWS:
 ☞ $\text{spec_rate} = \text{IF}(\text{Raw_Reqs} > 0)$
 THEN $\text{daily_MP_to_spec} * \text{spec_prod_rate} * \text{DT}$
 ELSE 0
 DOCUMENT:
 Requirements specification rate (i.e., requirements specified per day)
- OUTFLOWS:
 ☞ $\text{spec_QA_rate} = \text{Reqs_Spec} / (\text{average_QA_delay}/\text{DT}) * \text{daily_MP_to_spec_QA}/(\text{daily_MP_to_spec_QA} + 0.00001)$
 DOCUMENT:
 Number of requirements reviewed per day
- ☞ $\text{spec_deletion} = \text{spec_del_due_to_RC} + \text{spec_del_due_to_int}$
- $\text{Total_Raw_Requirements}(t) = \text{Total_Raw_Requirements}(t - dt) + (\text{raw_reqs_cum_rate} - \text{raw_reqs_del_rate}) * dt$
 INIT $\text{Total_Raw_Requirements} = \text{INIT}(\text{Raw_Reqs})$
 INFLOWS:
 ☞ $\text{raw_reqs_cum_rate} = \text{reqs_incoming_rate}$
- OUTFLOWS:
 ☞ $\text{raw_reqs_del_rate} = \text{raw_reqs_del_due_to_RC} + \text{raw_reqs_del_due_to_int}$
- $\text{average_QA_delay} = 10$
 DOCUMENT:
 Average delay for QA
 Set to 10 working days [7]
- $\text{frac_reqs_spec_QAed} = \text{Cum_QAed_Reqs_Spec}/(\text{Cum_Reqs_Spec} + 0.00001)$
- $\text{frac_spec_pcvd_completed} = \text{Cum_QAed_Reqs_Spec} / (\text{Pcvd_Project_Size} * 1000 / \text{LOC_per_reqs})$
- $\text{pcvd_reqs_phase_completed} = \text{IF}(\text{frac_reqs_spec_QAed} < 0.999 \text{ AND } \text{frac_reqs_spec_QAed} > 0.99 \text{ AND } \text{frac_daily_MP_to_reqs} > 0) \text{ THEN } 0 \text{ ELSE } 0$
- $\text{QAed_spec_to_dev_delay} = 1$
- $\text{spec_prod_rate} = (\text{spec_prod_ratio} * \text{actual_staff_prod_rate}) / \text{LOC_per_reqs}$
- $\text{spec_prod_ratio} = 55/7$
 DOCUMENT:
 Requirements specification ratio: the ratio of LOC per person-day and the number of requirements specified per person-day
 Set to 55/7 (calibrated against COCOMO 2.0, i.e., 55 person-months spent in programming and 7 person-months in requirements phase)

Staff Productive Time

- $Cum_Daily_Productive_Time(t) = Cum_Daily_Productive_Time(t - dt) + (DPT_change_rate) * dt$
 INIT $Cum_Daily_Productive_Time = 0$
 INFLOWS:
 ☞ $DPT_change_rate = average_productive_time$
- $Overtime(t) = Overtime(t - dt) + (overtime_incr_rate - overtime_decr_rate) * dt$
 INIT $Overtime = 0$
 INFLOWS:
 ☞ $overtime_incr_rate = IF (overtime_sought > Overtime)$
 THEN $(overtime_sought - Overtime) / (work_rate_adjustment_delay / DT)$
 ELSE 0
 OUTFLOWS:
 ☞ $overtime_decr_rate = IF (Overtime > overtime_sought)$
 THEN $(Overtime - overtime_sought) / (work_rate_adjustment_delay / DT)$
 ELSE 0
- $Project_Time(t) = Project_Time(t - dt) + (PT_inc_rate - PT_dec_rate) * dt$
 INIT $Project_Time = 0.75$
 INFLOWS:
 ☞ $PT_inc_rate = ST_dec_rate$
 OUTFLOWS:
 ☞ $PT_dec_rate = ST_inc_rate$
- $Slack_Time(t) = Slack_Time(t - dt) + (ST_inc_rate - ST_dec_rate) * dt$
 INIT $Slack_Time = 1 - iINIT(Project_Time)$
 INFLOWS:
 ☞ $ST_inc_rate = IF(indicated_slack_time > Slack_Time)$
 THEN $(indicated_slack_time - Slack_Time) / (work_rate_adjustment_delay / DT)$
 ELSE 0
 OUTFLOWS:
 ☞ $ST_dec_rate = IF (Slack_Time > indicated_slack_time)$
 THEN $(Slack_Time - indicated_slack_time) / (work_rate_adjustment_delay / DT)$
 ELSE 0
- $average_productive_time =$
 $(1 - overall_communication_overhead) * (Project_Time + effective_overtime - training_time)$
- $effective_overtime = Overtime * overtime_efficiency$
- $indicated_overwork_time = IF(overwork_duration < 1)$
 THEN 0
 ELSE IF $(MP_gap_handled > 0)$
 THEN $MP_gap_handled / (current_WF * overwork_duration + 0.00001)$
 ELSE IF $(MP_excess_absorbed > 0)$
 THEN $(0 - MP_excess_absorbed) / (current_WF * MAX(20, project_time_remaining) + 0.00001)$
 ELSE 0

- indicated_slack_time = IF (overwork_duration<1)
THEN 0.2
ELSE IF (indicated_overwork_time>Slack_Time-0.1)
THEN 0.1
ELSE IF (indicated_overwork_time>=0 AND indicated_overwork_time<Slack_Time-0.1)
THEN MAX(0.1, MIN(0.3, Slack_Time-indicated_overwork_time))
ELSE IF (indicated_overwork_time<0 AND 0-indicated_overwork_time<=average_productive_time)
THEN MAX(0.1, MIN(0.3, Slack_Time-indicated_overwork_time))
ELSE IF (0-indicated_overwork_time>average_productive_time)
THEN MAX(0.1, MIN(0.3, Slack_Time+average_productive_time))
ELSE 0.2
- new_staff_training_time = 0.6
- overtime_efficiency = 1
DOCUMENT:
Assumption: Project staff spends 100% of their overtime on the project.
- overtime_sought = IF(overwork_duration<1)
THEN 0
ELSE IF (indicated_overwork_time>indicated_slack_time)
THEN MIN(0.5, indicated_overwork_time-indicated_slack_time)
ELSE 0
- project_average_daily_productive_time = IF(TIME=0)
THEN 0
ELSE Cum_Daily_Productive_Time/TIME
- trainer's_time_per_new_staff = 0.2
DOCUMENT:
Assumptions:
1. Experienced staff spends 60% of their daily time on the project.
2. Each experienced staff can train three new staff members [AHM91].
Set at 0.2 (i.e., 0.6/3)
- training_time = SMTH1((new_staff_training_time*New_Staff+trainer's_time_per_new_staff*New_Staff)
/ (Exp_Staff+New_Staff), 5)
- work_rate_adjustment_delay = 10
DOCUMENT:
The average delay that project staff members adjust their work rate.
Set at 10 working days [AHM91]

Staff Productivity

- Exhaustion_Level(t) = Exhaustion_Level(t - dt) + (exh_buildup - exh_diminish) * dt
INIT Exhaustion_Level = 0
INFLOWS:
 - ☞ exh_buildup = IF (exh_diminish>0.0001 AND overwork_checkpoint=1)
THEN 0
ELSE exhaustion_inc_rate
- OUTFLOWS:

- $exh_diminish = IF(overwork_checkpoint=1 OR overwork=0)$
 THEN $Exhaustion_Level / (exh_diminish_time/DT)$
 ELSE 0
- $Max_Exh_Check_Point(t) = Max_Exh_Check_Point(t - dt) + (max_ECP_inc_rate - max_ECP_dec_rate) * dt$
 INIT $Max_Exh_Check_Point = 0$
 INFLOWS:
 $max_ECP_inc_rate = exh_buildup$
 OUTFLOWS:
 $max_ECP_dec_rate = IF(exh_diminish < 0.01 AND overwork_checkpoint=1)$
 THEN $PULSE(Max_Exh_Check_Point)$
 ELSE 0
- $actual_staff_prod_rate = nominal_staff_prod_rate * (SP_effect_on_prod_rate/1) * (exhaustion_effect_on_prod_rate/1) * (learning_effect_on_prod_rate/1)$
- $exh_diminish_time = 20$
 DOCUMENT:
 Set to 20 working days [7]
- $max_MP_shortage_to_be_handled = max_overwork_time * max_overwork_duration * current_WF$
- $max_overwork_duration = 50$
 DOCUMENT:
 The maximum duration that project staff members are willing to work overtime
 Set at 50 working days [7]
- $max_overwork_time = 0.6$
- $nominal_staff_prod_rate = frac_staff_exp * LOC_per_dev_unit + 0.5 * (1 - frac_staff_exp) * LOC_per_dev_unit$
 DOCUMENT:
 Set at one development units per day for the experienced staff members
 Set at 0.5 development units per day for new staff members [7]
- $overwork = MAX(0, Overtime + (INIT(Slack_Time) - Slack_Time))$
- $overwork_checkpoint = SWITCH(Max_Exh_Check_Point, 45)$
- $overwork_duration = IF(exh_diminish > 0.02)$
 THEN 0
 ELSE $max_overwork_duration * exhaustion_effect_on_overwork_duration$
- $overwork_willingness = IF(exh_diminish > 0.01)$
 THEN 0
 ELSE 1
- $exhaustion_effect_on_overwork_duration = GRAPH(Exhaustion_Level)$
 (0.00, 1.00), (10.0, 0.8), (20.0, 0.6), (30.0, 0.4), (40.0, 0.2), (50.0, 0.00)
 DOCUMENT:
 Adapted from the "Multiplier to the Overwork Duration Threshold due to Exhaustion" parameter [7]

- $\text{Defects_Found_in_SIT}(t) = \text{Defects_Found_in_SIT}(t - dt) + (\text{defects_detection_rate} - \text{defects_FIT_correction_rate}) * dt$
 INIT Defects_Found_in_SIT = 0
 INFLOWS:
 ☞ $\text{defects_detection_rate} = \text{num_of_defects_detected_per_unit} * \text{testing_rate} * \text{test_effectiveness}$
- OUTFLOWS:
 ☞ $\text{defects_FIT_correction_rate} = (\text{DT} * \text{daily_MP_to_defects_FIT_correction}) / (\text{effort_to_correct_a_defect_FIT} + 0.00001)$
- $\text{Defects_Released}(t) = \text{Defects_Released}(t - dt) + (\text{defects_releasing_rate}) * dt$
 INIT Defects_Released = 0
 INFLOWS:
 ☞ $\text{defects_releasing_rate} = \text{num_of_defects_detected_per_unit} * \text{testing_rate} * (1 - \text{test_effectiveness})$
- $\text{Integrated_Units}(t) = \text{Integrated_Units}(t - dt) + (\text{units_integration_rate} - \text{testing_rate}) * dt$
 INIT Integrated_Units = 0
 INFLOWS:
 ☞ $\text{units_integration_rate} = \text{SIT_degree_of_concurrency} * \text{daily_MP_to_integration} / (0.5 * \text{testing_effort_per_unit} + 0.00001)$
- OUTFLOWS:
 ☞ $\text{testing_rate} = \text{SIT_degree_of_concurrency} * (\text{DT} * \text{daily_MP_to_test}) / (0.5 * \text{testing_effort_per_unit} + 0.00001)$
- $\text{PreTest_Defects}(t) = \text{PreTest_Defects}(t - dt) + (\text{pretest_def_inc_rate}) * dt$
 INIT PreTest_Defects = 0
 INFLOWS:
 ☞ $\text{pretest_def_inc_rate} = \text{pretest_defects_inc_rate}$
- $\text{Units_To_Be_Integrated}(t) = \text{Units_To_Be_Integrated}(t - dt) + (\text{deved_units_inc_rate} - \text{units_integration_rate}) * dt$
 INIT Units_To_Be_Integrated = 0
 INFLOWS:
 ☞ $\text{deved_units_inc_rate} = \text{deved_units_incoming_rate}$
- OUTFLOWS:
 ☞ $\text{units_integration_rate} = \text{SIT_degree_of_concurrency} * \text{daily_MP_to_integration} / (0.5 * \text{testing_effort_per_unit} + 0.00001)$
- $\text{defects_removed} = \text{IF}(\text{Current_PreTest_Defects} < 1 \text{ AND } \text{Defects_Found_in_SIT} < 1) \text{ THEN } 1 \text{ ELSE } 0$
- $\text{deved_units_incoming_rate} = \text{QAed_deved_units_to_test}$
- $\text{effort_to_correct_a_defect_FIT} = 0.5$
- $\text{frac_units_integrated} = \text{Cum_Units_Integrated} / \text{pcvd_total_integrated_units}$
- $\text{frac_units_tested} = \text{Cum_Units_Tested} / (\text{pcvd_total_integrated_units} + 0.00001)$
- $\text{nom_testing_effort_per_unit} = 1.5$
- $\text{num_of_defects_detected_per_unit} = 1.2$
- $\text{pcvd_total_integrated_units} = \text{pcvd_total_dev_units}$

- $\text{project_pcvd_completed} = \text{IF}(\text{Units_To_Be_Integrated} < 0.1 \text{ AND } \text{frac_units_tested} > 0.999 \text{ AND } \text{Current_PreTest_Defects} < 0.25 \text{ AND } \text{Defects_Found_in_SIT} < 0.25)$
THEN PAUSE
ELSE 0
- $\text{SIT_degree_of_concurrency} = 1$
- $\text{testing_effort_per_unit} = \text{nom_testing_effort_per_unit} * \text{mult_to_testing_effort}$
- $\text{mult_to_testing_effort} = \text{GRAPH}(\text{number_of_teams})$
(1.00, 1.00), (2.00, 1.05), (3.00, 1.10), (4.00, 1.15), (5.00, 1.20), (6.00, 1.25), (7.00, 1.30), (8.00, 1.35), (9.00, 1.40), (10.0, 1.45)
- $\text{test_effectiveness} = \text{GRAPH}(\text{daily_MP_to_test})$
(0.00, 0.9), (1.00, 0.9), (2.00, 0.9), (3.00, 0.9), (4.00, 0.9), (5.00, 0.9), (6.00, 0.9), (7.00, 0.9), (8.00, 0.9), (9.00, 0.9), (10.0, 0.9)

System Integration and Test Manpower Allocation

- $\text{daily_MP_to_defects_FIT_correction} = \text{IF}(\text{frac_units_tested} < 1)$
THEN $\text{MIN}(\text{MP_needed_to_fix_a_defect_FIT} * \text{desired_defect_FIT_correction_rate}, \text{daily_MP_to_SIT_phase})$
ELSE $\text{daily_MP_to_SIT_phase}$
- $\text{daily_MP_to_integration} = \text{IF}(\text{frac_units_integrated} * 100 < 100)$
THEN $\text{MAX}(0, \text{daily_MP_to_SIT_phase} - \text{daily_MP_to_test} - \text{daily_MP_to_defects_FIT_correction})$
ELSE 0
- $\text{daily_MP_to_SIT_phase} = \text{IF}(\text{project_pcvd_completed} = 1)$
THEN 0
ELSE $\text{total_daily_MP} * \text{frac_daily_MP_to_SIT}$
- $\text{daily_MP_to_test} = \text{IF}(\text{frac_units_tested} < 1)$
THEN $(\text{daily_MP_to_SIT_phase} - \text{daily_MP_to_defects_FIT_correction}) * \text{frac_planned_SIT_MP_on_test}$
ELSE 0
- $\text{defects_FIT_correction_delay} = 5$
- $\text{desired_defect_FIT_correction_rate} = \text{Defects_Found_in_SIT} / (\text{defects_FIT_correction_delay})$
- $\text{frac_daily_MP_to_SIT} = 1 - \text{frac_daily_MP_to_reqs} - \text{frac_daily_MP_to_dev}$
- $\text{frac_planned_SIT_MP_on_test} = \text{IF}(\text{frac_units_tested} < 0.999)$
THEN 0.75
ELSE 0.75
- $\text{MP_needed_to_fix_a_defect_FIT} = \text{pcvd_def_FIT_correction_prod}$

System Integration and Test Manpower Needed

- $\text{Cum_Units_Integrated}(t) = \text{Cum_Units_Integrated}(t - dt) + (\text{integrated_units_cum_rate}) * dt$
INIT $\text{Cum_Units_Integrated} = 0$
INFLOWS:
 - $\text{integrated_units_cum_rate} = \text{units_integration_rate}$
- $\text{actual_def_FIT_correction_prod} = \text{IF}(\text{Defects_FIT_Corrected} > 0)$
THEN $\text{Defects_FIT_Corrected} / \text{Defects_FIT_Correction_Effort}$
ELSE $\text{planned_def_FIT_correct_prod}$

- $actual_intg_MP_needed = (pcvd_total_dev_units - Cum_Units_Integrated) / (actual_intg_prod + 0.000001)$
- $actual_intg_prod = IF(Cum_Units_Integrated > 0)$
THEN $Cum_Units_Integrated / (System_Integration_Effort + 0.00001)$
ELSE $planned_intg_prod$
- $actual_system_test_MP_need = (pcvd_total_dev_units - Cum_Units_Tested) / (actual_system_test_prod + 0.00001)$
- $actual_system_test_prod = IF(Cum_Units_Tested > 0)$
THEN $Cum_Units_Tested / System_Test_Effort$
ELSE $planned_system_test_prod$
- $current_planned_SIT_effort = init_planned_effort_to_SIT * (pcvd_total_dev_units / INIT(pcvd_total_dev_units))$
- $def_FIT_correction_effort_needed = Defects_Found_in_SIT * effort_to_correct_a_defect_FIT$
- $pcvd_def_FIT_correction_prod = SMTH1(actual_def_FIT_correction_prod, 20, actual_def_FIT_correction_prod)$
- $pcvd_SIT_effort_needed = weight_to_actual_SIT_MP_needed * (actual_intg_MP_needed + actual_system_test_MP_need + def_FIT_correction_effort_needed) + (1 - weight_to_actual_SIT_MP_needed) * SIT_effort_remaining$
- $planned_def_FIT_correct_prod = 5$
- $planned_intg_prod = pcvd_total_dev_units / (current_planned_SIT_effort * (1 - frac_planned_SIT_MP_on_test))$
- $planned_system_test_prod = pcvd_total_dev_units / (current_planned_SIT_effort * frac_planned_SIT_MP_on_test)$
- $SIT_effort_remaining = MAX(0, current_planned_SIT_effort - Cum_SIT_Effort)$
- $weight_to_actual_SIT_MP_needed = GRAPH(frac_units_tested)$
(0.00, 0.00), (0.1, 0.00), (0.2, 0.096), (0.3, 0.234), (0.4, 0.462), (0.5, 0.708), (0.6, 0.852), (0.7, 0.948), (0.8, 0.994), (0.9, 0.997), (1, 1.00)

Workforce

- $Desired_In_Trans_Staff(t) = Desired_In_Trans_Staff(t - dt) + (DITS_rate - new_staff_in_trans_rate) * dt$
INIT $Desired_In_Trans_Staff = 0$
INFLOWS:
 - $DITS_rate = staff_out_trans_rate$
 OUTFLOWS:
 - $new_staff_in_trans_rate = Desired_In_Trans_Staff / (in_trans_delay / DT)$
- $Exp_Staff(t) = Exp_Staff(t - dt) + (assimilation_rate - quit_rate - exp_staff_out_trans_rate) * dt$
INIT $Exp_Staff = initial_exp_WF$
INFLOWS:
 - $assimilation_rate = New_Staff / (assimilation_delay / DT)$
 OUTFLOWS:
 - $quit_rate = Exp_Staff / (employment_time / DT)$

- $\text{employment_time} = 673$
DOCUMENT:
Staff average employment time
Set at 673 working days [7]
- $\text{frac_staff_exp} = \text{Exp_Staff} / \text{current_WF}$
DOCUMENT:
Number of experienced staff divided by current total work force
- $\text{FTE_exp_staff} = \text{Exp_Staff} * \text{average_daily_MP_per_staff}$
- $\text{hiring_delay} = 40$
DOCUMENT:
Time to hire new project
Set at 40 working days [7]
- $\text{in_trans_delay} = 10$
DOCUMENT: Time to transfer staff into the project
Set at two weeks (i.e., ten working days) [7]
- $\text{max_new_staff} = \text{mx_new_hirees_per_exp_staff} * \text{FTE_exp_staff}$
- $\text{mx_new_hirees_per_exp_staff} = 3$
- $\text{out_trans_delay} = 10$
DOCUMENT:
Time to transfer staff out of the project
Set at 10 work days (same as the in-transfer delay)
- $\text{project_average_staff_level} = \text{IF}(\text{TIME} > 0)$
THEN $\text{Project_Staff_Level} / \text{TIME}$
ELSE 0
- $\text{staff_in_trans_rate} = \text{hiring_rate} + \text{new_staff_in_trans_rate}$
- $\text{staff_out_trans_rate} = \text{new_staff_out_trans_rate} + \text{exp_staff_out_trans_rate}$
- $\text{WF_production_delay} = \text{hiring_delay} + \text{assimilation_delay}$

APPENDIX C
KEY PROJECT STATISTICS OF THE EXAMPLE PROJECT

1. EXAMPLE is an organic-mode project.
2. Project real size is 64 KDSI (or $64000/60 = 1067$ tasks).
3. Project was underestimated by a factor of 1.5.
4. Initial estimate of the project size is 42.88 KDSI (or $2880/60 = 714.6$ tasks).
5. The distribution of effort expenditure is 22% for system testing, 15 to 20% for QA.
6. Staff's "actual productivity" is 33.84 DSI/man-day.
7. The "actual fraction of a man-day on project" is 60%.
8. Communication overhead is defined as a function of team size.
9. COCOMO's estimate for the average staffing level is 8 people. Therefore, communication overhead is around 5%.
10. Nominal staff productivity is 60 DSI/man-day. For the EXAMPLE project, a task is 60 DSI, and the nominal potential staff productivity is 1 task/man-day.
11. Average daily manpower per staff is 1 (i.e., staff work full-time on the EXAMPLE project).
12. The EXAMPLE project starts with a work force equal to half the "average staffing level," which is estimated to be eight project staff. Therefore, there are four project staff on board in the initial stage of the project.
13. Project took 430 work days to complete.

REFERENCES

- [1] K. R. Abbott, "Product development: a chunk at a time," Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering, IEEE Computer Society, Los Alamitos, CA, 1997.
- [2] T. K. Abdel-Hamid and S. E. Madnick, "A model of software project management dynamics," Proceedings of the 6th Annual International Computer Software and Applications Conference, IEEE Computer Society, 1982, pp. 539-54.
- [3] T. K. Abdel-Hamid and S. E. Madnick, "Impact of schedule estimation on software project behavior," *IEEE Software*, Vol. 3, No. 4, July 1986, pp. 70-75.
- [4] T. K. Abdel-Hamid, "The dynamics of software project staffing: a system dynamics based simulation approach," *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, February 1989, pp. 109-119.
- [5] T. K. Abdel-Hamid, "A study of staff turnover, acquisition, and assimilation and their impact on software development cost and schedule," *Journal of Management Information Systems*, Vol. 6, No. 1, 1989, pp. 21-40.
- [6] T. K. Abdel-Hamid, "Investigating the cost/schedule trade-off in software development," *IEEE Software*, Vol. 7, No. 1, January 1990, pp. 97-105.
- [7] T. K. Abdel-Hamid and S. E. Madnick, *Software project dynamics: an integrated approach*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [8] T. K. Abdel-Hamid, "Investigating the impacts of managerial Turnover/succession on software project performance," *Journal of Management Information Systems*, Vol. 9, No. 2, Fall 1992, pp. 127-144.

- [9] T. K. Abdel-Hamid, "Thinking in circles," *American Programmer*, Vol. 6, No. 5, May 1993, pp. 3-9.
- [10] T. K. Abdel-Hamid, "A multiproject perspective of single-project dynamics," *Journal of Systems and Software*, Vol. 22, No. 3, September 1993, pp. 151-165.
- [11] T. K. Abdel-Hamid, "The slippery path to productivity improvement," *IEEE Software*, Vol. 13, No. 4, July 1996, pp. 43-52.
- [12] M. Aoyama, "Concurrent development of software systems: a new development paradigm," *Software Engineering Notes*, Vol. 12, No. 3, July 1987, pp. 20-24.
- [13] M. Aoyama, "Distributed concurrent development of software systems: an object-oriented process model," *Proceedings of 14th Annual International Computer Software and Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 330-337.
- [14] M. Aoyama, "Concurrent development process model," *IEEE Software*, July 1993, pp. 46-55.
- [15] M. Aoyama, "Management of distributed concurrent development for large-scale software systems," *Proceedings of the 1995 Asia Pacific Software Engineering Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 158-167.
- [16] M. Aoyama, "Beyond software factories: concurrent-development process and an evolution of software process technology in Japan," *Information and Software Technology*, Vol. 38, 1996, pp. 133-143.
- [17] M. Aoyama, "Sharing the design information in a distributed concurrent development of large-scale software systems," *Proceedings of 20th Annual International Computer Software and Applications Conference*, pp. 168-175.

- [18] M. Aoyama, "Agile software process model," Proceedings 21st Annual International Computer Software and Applications Conference, IEEE Computer Society, Los Alamitos, CA, pp. 454-9.
- [19] M. Aoyama, "Managing the concurrent development of large-scale software systems," *International Journal of Technology Management*, Vol. 14, Nos 6/7/8, pp. 739-765.
- [20] M. Aoyama, interview with Aoyama and three other Fujitsu senior engineers, July 14-16, 1998.
- [21] J. D. Blackburn, G. Hoedemaker, and L. N. van Wassenhove, "Concurrent software engineering: prospects and pitfalls," *IEEE Transactions on Engineering Management*, Vol. 43, No. 2, May 1996, pp. 179-188.
- [22] B. Boehm, *Software engineering economics*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.
- [23] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, Vol. 1, 1995, pp. 45-60.
- [24] F. P. Brooks, Jr., *The mythical man-month*, Addison-Wesley, 1995.
- [25] P. G. Brown, "QFD: echoing the voice of the customer," *AT&T Technical Journal*, Vol. 70, No. 2, March-April 1991, pp. 18-32.
- [26] M. E. Bush and N.E. Fenton, "Software measurement: a conceptual framework," *Journal of Systems and Software*, Vol. 12, 1990, pp. 223-231.
- [27] CALS/Concurrent Engineering Task Group, "First principles of concurrent engineering: a competitive strategy for electronic system development," Review Draft, Washington D.C., CALS Industry Steering Group, 1991.

- [28] K. B. Clark and T. Fujimoto, "Overlapping problem solving in product development," in Ferdows, K., ed., *Managing International Manufacturing*, North-Holland, 1989, pp. 127-152.
- [29] Center for Software Engineering, "COCOMO II model definition manual: version 1.4," Computer Science Department, University of Southern California, <http://sunset.edu/Cocomo.html>, 1997.
- [30] K. G. Cooper, "The rework cycle: benchmarks for the project manager," *Project Management Journal*, Fall 1993, pp. 8-12.
- [31] K. G. Cooper and T. W. Mullen, "Swords and plowshares: the rework cycles of defense and commercial software development projects," *American Programmer*, Vol. 6. No. 5, May 1993, pp. 41-51.
- [32] M. I. Elboushi and J.S. Sherif, "Object-oriented software design utilizing Quality Function Deployment," *Journal of Systems and Software*, Vol. 38, No. 2, August 1997, pp. 133-143.
- [33] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, Vol. 31, No. 11, 1988, pp. 1268-1287.
- [34] M. A. Cusumano and R. W. Selby, *Microsoft secrets: how the world's most powerful software company creates technology, shapes markets, and manages people*, Free Press, 1995.
- [35] M. A. Cusumano and R. W. Selby, "How Microsoft builds software," *Communications of the ACM*, Vol. 40, No. 6, June 1997, pp. 53-61.
- [36] J. W. Forrester, *Industrial dynamics*, The MIT Press, Cambridge, MA, 1961.
- [37] L. Garber and D. Sims, "In pursuit of hardware-software codesign," *IEEE Computer*, June 1998, pp. 12-14.

- [38] R. L. Gordon and J. C. Lamb, "A close look at Brooks' Law," *Datamation*, June 1977, pp. 81-86.
- [39] S. Haag and P. Hogan, "Research issues in software quality function deployment: a new beginning for software engineering methodologies," Proceedings Decision Sciences Institute '92, DSI, Atlanta, GA, 1992, pp. 926-928.
- [40] S. Haag, M.K. Raja, and L.L. Schkade, "Quality function deployment usage in software development," *Communications of the ACM*, January 1996, Vol. 39, No. 1, pp. 41-49.
- [41] J. R. Hartley, *Concurrent engineering: shortening lead times, raising quality and lowering costs*, Cambridge, MA: Productivity Press, 1992.
- [42] K. Horner, "Methodology as a productivity tool," in *Software Engineering Productivity Handbook*, J. Keyes, ed., 1993, pp. 45-60.
- [43] *ithink analyst* Technical Documentation, High Performance Systems, Inc., 1996.
- [44] E. Jandourek, "A model for platform development," *Hewlett-Packard Journal*, August 1996.
- [45] C. Jones, *Programming productivity*, McGraw-Hill Book Co., New York, 1986.
- [46] K. Karoui, R. Dssouli, and O. Cherkaoui, "Specification transformations and design for testability," Proceedings of the 1996 IEEE Global Telecommunications Conference, Vol. 1, 1996, IEEE Computer Society, Piscataway, NJ, pp. 680-685.
- [47] J. C. Kelly, J. S. Sherif, and J. Hops, "An analysis of defect densities found during software inspections," *Journal of Systems and Software*, 17, 1992, pp. 111-117.

- [48] C. Y. Lin and R. R. Levary, "Computer-aided software development process design," *IEEE Transactions of Software Engineering*, Vol. 15, No. 9, September 1989, pp. 1025-1037.
- [49] C. Y. Lin, "Walking on battlefields: tools for strategic software management," *American Programmer*, Vol. 6. No. 5, May 1993, pp. 33-40.
- [50] C. Y. Lin, T. Abdel-Hamid, and J. S. Sherif, "Software-engineering process simulation model (SEPS)," *Journal of Systems and Software*, Vol. 38, 1997, pp. 263-277.
- [51] J. C. Lin, P. L. and S.C. Yang, "Promoting the software design for testability towards a partial test oracle," Proceedings of the 1997 8th IEEE International Workshop on Software Technology and Engineering Practice, STEP, 1997, IEEE Computer Society, Los Alamitos, CA, pp. 209-214.
- [52] R. Madachy, A software project dynamics model for project cost, schedule and risk assessment, Ph.D. dissertation, Department of Industrial and Systems Engineering, USC, December 1994.
- [53] R. J. Madachy, "System dynamics modeling of an inspection-based process," Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 376-386.
- [54] J. Martin, *Rapid application development*, MacMillan, 1991.
- [55] J. McCarthy, *Dynamics of software development*, Microsoft Press, 1995.
- [56] S. C. McConnell, *Code complete: a practical handbook of software construction*, Microsoft Press, Redmond, WA, 1993.
- [57] S. C. McConnell, *Rapid development: taming wild software schedules*, Microsoft Press, Redmond, WA, 1996.

- [58] G. J. Myers, "A controlled experiment in program testing and code walk through/inspections," *Communications of the ACM*, Vol. 21, No. 9, September 1978, pp. 760-768.
- [59] R. J. Muller, *Productive objects-an applied software project management framework*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.
- [60] B. Prasad, *Concurrent engineering fundamentals*, Prentice Hall, PTR, 1996.
- [61] R. S. Pressman, *Software engineering; a practitioner's approach*, 4th edition, McGraw-Hill, 1997.
- [62] P. Pulli and R. Elmstrom, "IPTES: a concurrent engineering approach for real-time software development," *Real-Time Systems*, Vol. 5, 1993, pp. 139-152.
- [63] P. J. Pulli and M. P. Heikkinen, "Concurrent engineering for real-time systems," *IEEE Software*, Vol. 10, Nov. 1993, pp. 39-44.
- [64] L. H. Putnam and W. Myers, *Industrial strength software: effective management using measurement*, IEEE Computer Society Press, Los Alamitos, CA, 1997.
- [65] F. Rafii and S. Perkins, "Internationalizing software with concurrent engineering," *IEEE Software*, Vol. 12, No. 5, September 1995, pp. 39-46.
- [66] M. Ramachandran and W. Fleischer, "Design for large scale software reuse: an industrial case study," *Proceedings of the 4th International Conference on Software Reuse*, IEEE Computer Society, Los Alamitos, CA, 1996, pp. 104-111.
- [67] A. Rosenblatt and G. F. Watson, "Concurrent engineering," *IEEE Spectrum*, July 1991, pp. 22-37.
- [68] A. Rodrigues and T. Williams, "System dynamics in software project management: towards the development of a formal integrated framework," Pro-

- ceeding of the 1996 International System Dynamics Conference, July 21-25, Cambridge, MA.
- [69] S. G. Shina, *Concurrent engineering and design for manufacture of electronic products*, New York: Van Nostrand Reinhold, 1991.
- [70] R. Thackeray and G. van Treeck, "Applying quality function deployment for software product development," *Journal of Engineering Design*, Vol. 1, No. 4, 1990, pp. 389-410.
- [71] T. L. Tran, "QFD application to a software-intensive system development project," *Proceedings of the 1996 IEEE International Engineering Management Conference*, IEEE Piscataway, NJ, pp. 683-689.
- [72] T. L. Tran and J.S. Sherif, "Quality function deployment (QFD): an effective technique for requirements acquisition and reuse," *Proceedings of the 2nd IEEE International Software Engineering Standards Symposium*, IEEE Computer Society, Los Alamitos, CA, pp. 191-200.
- [73] H. T. Yeh, "Re-engineering a software development process for fast delivery-approach & experiences," *Proceedings of the First International Conference on the Software Process*, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 106-112.
- [74] R. T. Yeh, "Notes on concurrent engineering," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 5, October 1992, pp. 407-414.
- [75] D. B. Simmons, N.C. Ellis, H. Fujihara, and W. Kuo, *Software measurement: A visualization toolkit for project control and process improvement*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1998.
- [76] R. P. Smith, "The historical roots of concurrent engineering fundamentals," *IEEE Transactions on Engineering Management*, Vol. 44, No. 1, February 1997, pp. 67-78.

- [77] P. J. Starr, "Modeling issues and decisions in system dynamics," *TIMS Studies in the Management Science*, Vol. 14, 1980, pp. 45-59.
- [78] J. L. Turino, *Managing concurrent engineering: buying time to a market: a definitive guide to improved competitiveness in electronics design and manufacturing*, New York: Van Nostrand Reinhold, 1992.
- [79] J. D. Tvedt and J. S. Collofello, "Evaluating the effectiveness of process improvements on software development cycle time via system dynamics modeling," Proceedings of the 19th Annual International Computer Software and Applications Conference, 1995, pp. 318-325.
- [80] J. D. Tvedt, *An extensible model for evaluating the impact of process improvements on software development cycle time*, Ph.D. dissertation, Arizona State University, May 1996.
- [81] D. M. Weiss, "Evaluating software development by error analysis," *Journal of Systems and Software*, Vol. 1, 1979, pp. 57-70.
- [82] N. Whitten, *Managing software development process: formula for success*, second edition, John Wiley & Sons, Inc., 1995.
- [83] H. P. E. Vranken, M. F. Witteman, and R. C. van Wuijtswinkel, "Design for testability in hardware-software systems," *IEEE Design & Test of Computers*, Vol. 13, No. 3, Fall 1996, pp. 79-87.
- [84] G. M. Weinberg, *Quality software management: Volume 1, system thinking*, Dorset House Publishing, 1992.
- [85] R. I. Winner, J. P. Pennell, H. E. Bertrend, and M. M. G. Slusarczuk, The role of concurrent engineering in weapons system acquisition. *IDA Report R-338*, Alexandria, VA: Institute for Defense Analyses, 1988.

- [86] B. J. Zirger and J. L. Hartley, "The effect of acceleration techniques on product development time," *IEEE Transactions on Engineering Management*, Vol. 43, No. 2, May 1996, pp. 143-152.
- [87] R. E. Zultner, "Software quality [function] deployment," *ASQC Quality Congress Transactions*, 1989, pp. 558-563.

BIOGRAPHICAL INFORMATION

Chih-tung Hsu received the degree of Bachelor of Science in Civil Engineering from National Chiao Tung University, Taiwan, in 1986, the degree of Masters of Science in Mechanical Engineering from Tamkang University, Taiwan, in 1988, the degree of Masters of Science in Computer Science and Engineering from The University of Texas at Arlington on December 1992, and the degree of Doctor of Philosophy in Computer Science and Engineering from The University of Texas at Arlington in 1999.

He received an outstanding research by a Ph.D. student award in 1998. He has published eight technical papers in the areas of software incremental delivery, object-oriented development and testing methodologies, and system dynamics modeling during his Ph.D study.

He taught classes while he was working for his Ph.D. degree, including software engineering, object-oriented software engineering, algorithms and data structures. His research interests include system dynamics software process modeling, techniques for software requirements specification, and methodologies for object-oriented software development and testing.

