

PHD DISSERTATION

A MULTI-LEVEL CODE COMPREHENSION MODEL FOR LARGE SCALE  
SOFTWARE

Submitted by

A. Marie Vans

Department of Computer Science

In partial fulfillment of the requirements  
for the degree of Doctor Of Philosophy  
Colorado State University  
Fort Collins, Colorado  
Fall 1996



U18402 0346866

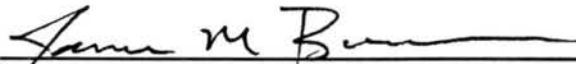
S64  
#357  
1996

COLORADO STATE UNIVERSITY

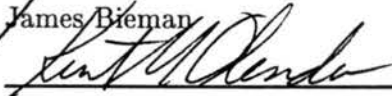
November 12, 1996

WE HEREBY RECOMMEND THAT THE PHD Dissertation PREPARED UNDER OUR SUPERVISION BY A. Marie Vans ENTITLED A Multi-level Code Comprehension Model For Large Scale Software BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF Doctor Of Philosophy.

Committee on Graduate Work



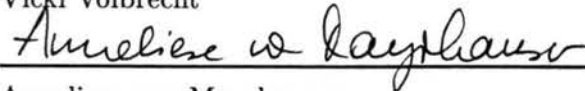
James Bieman



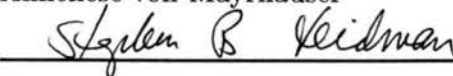
Kurt Olender



Vicki Volbrecht



Anneliese von Mayrhauser



Department Head

COLORADO STATE UNIVERSITY LIBRARIES

## ABSTRACT OF PHD DISSERTATION

### A MULTI-LEVEL CODE COMPREHENSION MODEL FOR LARGE SCALE SOFTWARE

For the past 20 years researchers have studied how programmers understand code they did not write. Most of this research has concentrated on small-scale code understanding. We consider it necessary to design studies that observe programmers working on large-scale code in production environments. We describe the design and implementation of such a study which included 11 maintenance engineers working on various maintenance tasks. The objective is to build a theory based on observations of programmers working on real tasks. Results show that programmers understand code at different levels of abstraction. Expertise in the application domain, amount of prior experience with the code, and task can determine the types of actions taken during maintenance, the level of abstraction at which the programmer works, and the information needed to complete a maintenance task. A better grasp of how programmers understand large scale code and what is most efficient and effective can lead to better tools, better maintenance guidelines, and documentation.

A. Marie Vans  
Department of Computer Science  
Colorado State University  
Fort Collins, Colorado 80523  
Fall 1996

## ACKNOWLEDGEMENTS

I wish to extend my thanks first and foremost to my adviser, Dr. Anneliese von Mayrhauser. Without her knowledge, experience, guidance, and support none of this work would have been possible. I also thank Dr. James Bieman for his contributions and for guidance in the area of experimental design. I am grateful to Dr. Kurt Olender and Dr. Vicki Volbrecht for their contributions. I would also like to thank Nancy Pennington for her guidance on the model specification and validation. Last but certainly not least, I thank Dr. V. Scott Gordon for his endless encouragement, support, and friendship.

## DEDICATION

This dissertation is dedicated to the strong women in my life who, through perseverance, encouragement, wisdom, and example, made it possible for me to complete this work. The women without whom I would never have succeeded: my grandmother: Agnes Jackson, my mother: Mariann Jackson, my sister: Lisa Vans Kreskey, and of course my advisor: Anneliese von Mayrhauser.

## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The Problem</b>	<b>4</b>
2.1	Rationale . . . . .	4
2.2	The Problem Statement . . . . .	13
2.3	Elements of the Problem . . . . .	13
2.4	Scope of the Study . . . . .	14
<b>3</b>	<b>Background</b>	<b>15</b>
3.1	Overview . . . . .	15
3.2	Common Elements of Cognition Models . . . . .	15
3.2.1	Knowledge . . . . .	15
3.2.2	Mental Model . . . . .	16
3.2.3	Hypotheses . . . . .	18
3.2.4	Strategies . . . . .	19
3.2.5	Facilitating Knowledge Acquisition . . . . .	20
3.2.6	Expert Characteristics . . . . .	21
3.3	Models . . . . .	23
3.3.1	Letovsky Model . . . . .	23
3.3.2	Shneiderman Model . . . . .	25
3.3.3	Brooks Model . . . . .	27
3.3.4	Top-Down – Soloway & Ehrlich . . . . .	30
3.3.5	Pennington’s Model – Bottom-Up Comprehension . . . . .	33
3.3.6	Evaluation of Models . . . . .	37
3.4	Integrated Model . . . . .	38
3.5	Levels of the Integrated Model . . . . .	39
3.6	Research Grid . . . . .	41
<b>4</b>	<b>Research Design</b>	<b>44</b>
4.1	Experimental Design . . . . .	44
4.2	Program Comprehension Study Objectives . . . . .	47
4.3	Protocol Analysis . . . . .	48
4.4	Enumeration and Segmentation of Action Types – Level 3 . . . . .	51
4.5	Process Discovery – Level 2 & Level 1 . . . . .	52
4.6	Action Type Analysis: Hypotheses & Switching Between Models . . . . .	53
4.6.1	Hypotheses – Level 1 . . . . .	54
4.6.2	Switching between Model Components – Model Level . . . . .	55
4.7	Strategy Identification – Level 1 . . . . .	55
4.8	Information Needs . . . . .	56
4.9	Summary . . . . .	57

<b>5 Results</b>	<b>58</b>
5.1 Actions – Level 3 . . . . .	58
5.1.1 Action Frequencies Across All Tasks . . . . .	59
5.1.2 Actions – Model Frequencies by Task, Expertise, & Accumulated Knowledge . . . . .	62
5.2 Processes – Levels 2 & 1 . . . . .	69
5.2.1 Systematic Processes . . . . .	69
5.2.2 Opportunistic Processes – Level 1 . . . . .	76
5.2.3 Combined Systematic & Opportunistic Processes . . . . .	86
5.2.4 Interpretation and Conclusions on Comprehension Processes . . . . .	87
5.3 Hypotheses – Level 1 . . . . .	88
5.3.1 Hypothesis Type: Overall Frequencies & Model Frequencies by Task, Expertise, & Accumulated Knowledge . . . . .	89
5.3.2 Hypothesis Generated Switching Behavior . . . . .	98
5.3.3 HK-Ratio – Using Hypotheses and Knowledge to Indicate Expertise . . . . .	103
5.4 The Role of Model Components in the Integrated Model – Level 1 . . . . .	107
5.4.1 Switches Between Models . . . . .	108
5.4.2 The Role of Code Size on Model Building . . . . .	114
5.5 Information Needs . . . . .	114
5.5.1 Frequencies for All Subjects . . . . .	115
5.5.2 Information Needs – By Task . . . . .	120
5.5.3 Information Needs – By Expertise . . . . .	121
5.5.4 Information Needs – By Accumulated Knowledge . . . . .	122
5.6 Tool Capabilities . . . . .	122
5.7 Core Competencies . . . . .	126
5.8 Conclusions . . . . .	129
5.8.1 Actions . . . . .	129
5.8.2 Processes . . . . .	129
5.8.3 Hypotheses . . . . .	130
5.8.4 Switching Between Models – Action Generated . . . . .	131
5.8.5 Role of Code Size . . . . .	132
5.8.6 Information Needs and Tool Capabilities . . . . .	132
<b>6 Future Directions</b>	<b>134</b>
6.1 Integrated Model Construction . . . . .	134
6.2 Hypothesis Generation . . . . .	136
6.3 Switching Behavior . . . . .	137
6.4 Information Needs & Tool Capabilities . . . . .	137
6.5 Conclusions . . . . .	139
<b>7 REFERENCES</b>	<b>142</b>
<b>A Classification of Model Components – Task Codes</b>	<b>148</b>
A.1 Top-Down Component Identification . . . . .	148
A.2 Program Model Component Identification . . . . .	152
A.3 Situation Model Component Identification . . . . .	156

<b>B Supplemental Data for Results</b>	<b>159</b>
B.1 Actions	159
B.1.1 Action Percentages By Task	159
B.1.2 Action Percentages By Expertise	163
B.1.3 Action Percentages By Accumulated Knowledge	165
B.2 Hypotheses – Level 1	168
B.2.1 Model Frequencies by Task, Expertise, & Accumulated Knowledge	169
B.2.2 Hypothesis Generated Switching Behavior	194
B.2.3 HK-Ratio – Using Hypotheses and Knowledge to Indicate Expertise	203
B.3 The Role of Model Components in the Integrated Model – Level 1	206
B.3.1 Switches Between Models	206
B.4 Information Needs	212
B.4.1 Information Needs – By Task	213
B.4.2 Information Needs – By Expertise	216
B.4.3 Information Needs – By Accumulated Knowledge	219
<b>C Programming Sessions – Individual Subject Results</b>	<b>223</b>
C.1 General Understanding	224
C.1.1 G1: Domain/Language Expert – General Understanding	225
C.1.2 G2: Domain/Language Expert – Understand One Module	229
C.2 Corrective Maintenance	257
C.2.1 C1: Domain Novice/Language Expert – Fix Reported Bug	257
C.2.2 C2: Domain Expert/Language Novice – Understand Bug	262
C.2.3 C3: Domain Expert/Language Novice – Fix Reported Bug	268
C.2.4 C4: Domain Expert/Language Novice – Track Down Bug	273
C.3 Enhancement	278
C.3.1 EN1: Domain/Language Expert – Add Functionality	278
C.3.2 EN2: Language/Domain Expert Add Function	283
C.4 Adaptation	288
C.4.1 AD1: Language/Domain Expert Port Program	289
C.4.2 AD2: Language Novice/Domain Expert Add Function	307
C.5 Code Leverage	313
C.5.1 L1: Language Expert/Domain Novice – Leverage Program	313



## LIST OF FIGURES

3.1	Letovsky – Comprehension Model . . . . .	24
3.2	Shneiderman – Comprehension Model . . . . .	26
3.3	Brooks – Comprehension Model . . . . .	29
3.4	Soloway & Ehrlich – Comprehension Model . . . . .	32
3.5	Pennington – Comprehension Model . . . . .	36
3.6	Integrated Code Comprehension Model . . . . .	40
4.1	Verbalization Task . . . . .	50
5.1	Systematic Processes: Episodic Process – Read Block . . . . .	70
5.2	Systematic Processes: Process PC – Aggregate-Level . . . . .	73
5.3	Process ABC – Session Level . . . . .	75
5.4	Opportunistic Processes: Goals 1 and 2 . . . . .	79
5.5	Opportunistic Processes: Goal 4 . . . . .	80
5.6	Opportunistic Processes: Goal 7 . . . . .	82
5.7	Opportunistic Processes: Goal Completion . . . . .	85
B.1	Adaptive Maintenance Actions By Model (% of total actions, 2 Subjects) . . . . .	160
B.2	Corrective Maintenance Actions By Model (% of total actions, 4 Subjects) . . . . .	161
B.3	Enhancement Maintenance Actions By Model (% of total actions, 2 Subjects) . . . . .	161
B.4	General Understanding Actions By Model (% of total actions, 3 Subjects) . . . . .	162
B.5	Language Only Expertise By Model (% of total actions, 2 Subjects) . . . . .	163
B.6	Domain Only Expertise By Model (% of total actions, 4 Subjects) . . . . .	164
B.7	Domain & Language Expertise By Model (% of total actions, 5 Subjects) . . . . .	165
B.8	Little Accumulated Knowledge By Model (% of total actions, 4 Subjects) . . . . .	166
B.9	Some Accumulated Knowledge By Model (% of total actions, 4 Subjects) . . . . .	166
B.10	Significant Accumulated Knowledge By Model (% of total actions, 3 Subjects) . . . . .	167
C.1	G1: General Understanding – Action Sequence . . . . .	225
C.2	G1: Hypotheses Sequence . . . . .	227
C.3	G2: Understand One Module – Action Sequence . . . . .	231
C.4	G2: Understand One Module – Hypotheses Sequence . . . . .	231
C.5	G2: Process 1 – Read Block . . . . .	235
C.6	G2: Process 2 – Integrate Partially Understood . . . . .	237
C.7	G2: Process 3 – Determine Var Definition . . . . .	239
C.8	G2: Process 4 – Incorporate Acquired Knowledge . . . . .	240
C.9	G2: Process 5 – Identify Block Boundaries . . . . .	243
C.10	G2: Process 6 – Resolve Deferred Questions . . . . .	245
C.11	G2: Process 7 – Understand Procedure Call . . . . .	247
C.12	G2: Process A – Aggregate-Level . . . . .	250

C.13 G2: Process B – Aggregate-Level . . . . .	251
C.14 G2: Process C – Aggregate-Level . . . . .	253
C.15 G2: Process ABC – Session Level . . . . .	256
C.16 C1: Fix Reported Bug – Action Sequence . . . . .	258
C.17 C1: Fix Reported Bug – Hypotheses Sequence . . . . .	258
C.18 C2: Understand Bug – Action Sequence . . . . .	262
C.19 C2: Fix Reported Bug – Hypotheses Sequence . . . . .	266
C.20 C3: Fix Reported Bug – Action Sequence . . . . .	268
C.21 C3: Fix Reported Bug – Hypotheses Sequence . . . . .	271
C.22 C4: Track Down Bug – Action Sequence . . . . .	274
C.23 C4: Track Down Bug – Hypotheses Sequence . . . . .	277
C.24 EN1: Enhancement – Action Sequence . . . . .	280
C.25 EN1: Enhancement – Hypotheses Sequence . . . . .	282
C.26 EN2: Enhancement – Action Sequence . . . . .	284
C.27 EN2: Hypotheses Sequence . . . . .	285
C.28 AD1: Port Programs – Action Sequence . . . . .	289
C.29 AD1:Port Programs – Hypotheses Sequence . . . . .	292
C.30 AD1: Goals 1 and 2 . . . . .	294
C.31 AD1: Goal 4 . . . . .	295
C.32 AD1: Goal 7 . . . . .	298
C.33 AD1: Goals 9 & 11 . . . . .	300
C.34 AD1: Goals 13 & 14 . . . . .	302
C.35 AD1: Goal 20 . . . . .	304
C.36 AD1: Goal Completion . . . . .	306
C.37 AD2: Port Program/Add Function – Action Sequence . . . . .	307
C.38 AD2: Hypotheses Sequence . . . . .	311
C.39 L1: Leverage Program – Action Sequence . . . . .	314
C.40 L1: Leverage Program – Hypotheses Sequence . . . . .	314

## LIST OF TABLES

2.1	Maintenance Life-cycle Subtasks . . . . .	5
2.2	Comprehension Experiments . . . . .	9
2.3	Experimental Purposes & References . . . . .	10
2.4	Experimental Purposes & References (continued) . . . . .	11
3.1	Code Comprehension Model – Levels of Detail . . . . .	39
3.2	Research Grid . . . . .	42
4.1	Programming Sessions – All Maintenance Tasks . . . . .	46
4.2	Protocol Analysis Steps . . . . .	51
4.3	Example Protocol Analysis – Action Types . . . . .	52
4.4	Example Protocol Analysis – Hypotheses . . . . .	54
4.5	Example Protocol Analysis – Information Needs . . . . .	57
5.1	Action Counts Across Tasks – Top-Down Model . . . . .	59
5.2	Action Counts Across Tasks – Program Model . . . . .	61
5.3	Action Counts Across Tasks – Situation Model . . . . .	62
5.4	Action-Types by Model – Frequencies & Percentages . . . . .	64
5.5	Expertise: Top-Down & Program/Situation Model Relative Frequencies . . . . .	67
5.6	Accumulated Knowledge: Top-Down & Program/Situation Model Relative Frequencies . . . . .	68
5.7	Episodic Process Frequency Count (Subject: G2) . . . . .	71
5.8	Understanding One Module – Information Needs for Process P1 . . . . .	72
5.9	Aggregate Processes – Episodic Composition . . . . .	73
5.10	Aggregate Process Frequency Count (Subject: G2) . . . . .	74
5.11	Process Trigger Frequencies (Subject: G2) . . . . .	74
5.12	Hypothesis-Type Frequencies – All Tasks . . . . .	91
5.13	Task Hypotheses by Model–Total & Per Subject Counts with Total Frequencies	93
5.14	Expert Hypotheses by Model–Total & Per Subject Counts with Total Frequencies	95
5.15	Accumulated Knowledge: Hypotheses by Model–Total & Per Subject Counts with Total Frequencies . . . . .	97
5.16	Hypotheses Switching Behavior – By Model Component for all Subjects (11) .	99
5.17	Hypotheses Switching Behavior – By Task . . . . .	100
5.18	Hypotheses Switching Behavior – Expertise . . . . .	101
5.19	Hypotheses Switching Behavior – Accumulated Knowledge . . . . .	103
5.20	HK-Ratio by Subject . . . . .	105
5.21	Overall HK-Ratio Rankings . . . . .	106
5.22	Action Switches – Absolute & Percent of Total Switches by Task & Expertise .	109
5.23	Action Switches – Absolute & Percent of Total Switches by Task . . . . .	110
5.24	Action Switches – Absolute & Percent of Total Switches by Expertise . . . . .	111

5.25	Action Switches – Absolute & Percent of Total Switches by Accumulated- Knowledge . . . . .	113
5.26	Frequencies of References to Model Components (All Subjects) . . . . .	115
5.27	Information Needs Frequencies: All Subjects . . . . .	117
5.28	Information Needs Frequencies: All Subjects (continued) . . . . .	118
5.29	Information Needs Frequencies: All Subjects (continued) . . . . .	119
5.30	Tool Capabilities Table . . . . .	125
5.31	Core Competencies by Model Component . . . . .	128
B.1	Labels for Figure – Domain Only Expertise . . . . .	164
B.2	Hypotheses by Model – Frequencies & Percentages . . . . .	168
B.3	Hypothesis-Type Frequencies/Percentages – Top-Down Model, All Tasks . . . . .	170
B.4	Hypothesis-Type Frequencies/Percentages – Program Model, All Tasks . . . . .	173
B.5	Hypothesis-Type Frequencies/Percentages – Situation Model, All Tasks . . . . .	176
B.6	Hypothesis-Type Frequencies/Percentages – Top-Down Model, Expertise . . . . .	178
B.7	Hypothesis-Type Frequencies/Percentages – Program Model, Expertise . . . . .	182
B.8	Hypothesis-Type Frequencies/Percentages – Situation Model, Expertise . . . . .	183
B.9	Hypothesis-Type Frequencies/Percentages – Top-Down Model, Accumulated Knowledge . . . . .	188
B.10	Hypothesis-Type Frequencies/Percentages – Program Model, Accumulated Knowledge . . . . .	191
B.11	Hypothesis-Type Frequencies/Percentages – Situation Model, Accumulated Knowledge . . . . .	192
B.12	Hypotheses Switching Behavior – Adaptation Task (AD1, AD2) . . . . .	194
B.13	Hypotheses Switching Behavior – Corrective Task (C1,C2,C3,C4) . . . . .	195
B.14	Hypotheses Switching Behavior – Enhancement Task (EN1,EN2) . . . . .	196
B.15	Hypotheses Switching Behavior – General Understanding Task (G1,G2,L1) . . . . .	197
B.16	Hypotheses Switching Behavior – Language Expertise (C1,L1) . . . . .	198
B.17	Hypotheses Switching Behavior – Domain Expertise (AD2,C2,C3,C4) . . . . .	199
B.18	Hypotheses Switching Behavior – Language & Domain Expertise (AD1,EN1,EN2,G1,G2) . . . . .	199
B.19	Hypotheses Switching Behavior – Little Accumulated Knowledge (C2,C3,EN2,G1) . . . . .	200
B.20	Hypotheses Switching Behavior – Some Accumulated Knowledge (C1,C4,G2,L1)	201
B.21	Hypotheses Switching Behavior – Some Accum. Knowledge (3 Subjects w/out G2) . . . . .	201
B.22	Hypotheses Switching Behavior – Significant Accumulated Knowledge (AD1,AD2,EN1) . . . . .	202
B.23	Program Model: HK-Ratio Rankings . . . . .	203
B.24	Situation Model: HK-Ratio Rankings . . . . .	204
B.25	Top-Down Model: HK-Ratio Rankings . . . . .	204
B.26	Combined Program & Situation Model: HK-Ratio Rankings . . . . .	205
B.27	Overall HK-Ratio Rankings . . . . .	205
B.28	Action Switches – Absolute & Percent of Total Switches by Task . . . . .	207
B.29	Action Switches – Absolute & Percent of Total Switches by Expertise . . . . .	209
B.30	Action Switches – Absolute & Percent of Total Switches by Accumulated Knowledge . . . . .	211
B.31	Information Needs Frequency By Task . . . . .	215

B.32	Information Needs Frequency By Task (continued)	216
B.33	Information Needs Frequency By Expertise	218
B.34	Information Needs Frequency By Expertise (continued)	219
B.35	Information Needs Frequency By Accumulated Knowledge	221
B.36	Information Needs Frequency By Accumulated Knowledge (continued)	222
C.1	G1: References and Action Switches Between Models	225
C.2	G1: Action-Type Frequencies – General Understanding	226
C.3	G1: Hypothesis Generated Switches Between Models	226
C.4	G1: Hypothesis-Type Frequencies – General Understanding	227
C.5	G1: General Understanding - Information Needs	228
C.6	G1: General Understanding – Information Needs (continued)	229
C.7	G2: Action-Type Frequencies – General Understanding	230
C.8	G2: References and Action Switches Between Models	230
C.9	G2: Hypothesis-Type Frequencies – Understand One Module	232
C.10	G2: Hypothesis Generated Switches Between Models	232
C.11	G2: Understand One Module - Information Needs	233
C.12	G2: Process Frequency Count	234
C.13	G2: Understanding One Module – Information Needs for Processes P1 – P7	248
C.14	G2: Processes A,B,C, and Aggregate Trigger Frequencies	254
C.15	C1: References and Action Switches Between Models	257
C.16	C1: Action-Type Frequencies – Fix Reported Bug	259
C.17	C1: Hypothesis Generated Switches Between Models	259
C.18	C1: Hypothesis-Type Frequencies – Fix Reported Bug	260
C.19	C1: Fix Reported Bug - Information Needs	261
C.20	C2: References and Action Switches Between Models	263
C.21	C2: Action-Type Frequencies – Understand Bug	264
C.22	C2: Hypothesis-Type Frequencies – Understand Bug	265
C.23	C2: Hypothesis Generated Switches Between Models	265
C.24	C2: Fix Reported Bug - Information Needs	267
C.25	C3: Action-Type Frequencies – Fix Reported Bug	269
C.26	C3: References and Action Switches Between Models	269
C.27	C3: Hypothesis-Type Frequencies – Fix Reported Bug	270
C.28	C3: Hypothesis Generated Switches Between Models	270
C.29	C3: Fix Reported Bug - Information Needs	272
C.30	C4: References and Action Switches Between Models	274
C.31	C4: Action-Type Frequencies – Track Down Bug	275
C.32	C4: Hypothesis-Type Frequencies – Track Down Bug	276
C.33	C4: Hypothesis Generated Switches Between Models	277
C.34	C4: Track Down Bug - Information Needs	278
C.35	EN1: Action-Type Frequencies – Enhancement	279
C.36	EN1: References and Action Switches Between Models	280
C.37	EN1: Hypothesis-Type Frequencies – Enhancement	281
C.38	EN1: Hypothesis Generated Switches Between Models	281
C.39	EN1: Enhancement - Information Needs	282
C.40	EN2: References and Action Switches Between Models	285
C.41	EN2: Action-Type Frequencies – Enhancement	286
C.42	EN2: Hypothesis-Type Frequencies Enhancement	287

C.43 EN2: Hypothesis Generated Switches Between Models . . . . .	287
C.44 EN2: Enhancement - Information Needs . . . . .	288
C.45 AD1: Action-Type Frequencies – Port Programs . . . . .	290
C.46 AD1: References and Action Switches Between Models . . . . .	290
C.47 AD1: Hypothesis-Type Frequencies – Port Programs . . . . .	291
C.48 AD1: Hypothesis Generated Switches Between Models . . . . .	291
C.49 AD1: Port Programs - Information Needs . . . . .	292
C.50 AD2: Action-Type Frequencies – Port Program/Add Function . . . . .	308
C.51 AD2: References and Action Switches Between Models . . . . .	309
C.52 AD2: Hypothesis-Type Frequencies – Port Program/Add Function . . . . .	310
C.53 AD2: Hypothesis Generated Switches Between Models . . . . .	310
C.54 AD2: Port Program/Add Function - Information Needs . . . . .	312
C.55 L1: Action-Type Frequencies – Leverage Program . . . . .	315
C.56 L1: References and Action Switches Between Models . . . . .	315
C.57 L1: Hypothesis-Type Frequencies – Leverage Program . . . . .	316
C.58 L1: Hypothesis Generated Switches Between Models . . . . .	316
C.59 L1: Leverage Program - Information Needs . . . . .	317

## Chapter 1

### INTRODUCTION

For the past 20 years researchers have studied how programmers understand code they did not write. Code understanding is common, especially for maintenance engineers. Maintenance engineers assume responsibility for enhancing, fixing defects, and adapting very large systems which they may never completely understand. Often, maintenance programmers must make changes to the code they do not understand. Other times no changes can be made until the code is reasonably understood. In either case, the cost of maintaining large-scale code can be high. Realizing how programmers actually comprehend programs helps to define better tools, guidelines, and documentation.

Unfortunately, much of the research focuses on very small aspects of the program comprehension process and its contribution to large-scale code understanding is not clear. Many of these small-scale, controlled experiments make assumptions about how comprehension proceeds. Frequently, these assumptions are not based on a well defined theory. In contrast, we observed programmers working on large-scale software during actual maintenance tasks. Such an effort contributes to building a theory of program comprehension. Once a theory of large-scale program comprehension has been developed, further controlled experiments will be appropriate for validating it. There is a tradeoff between the experimental control associated with experiments that make use of small-scale code and the generality gained with studies that use large-scale code from realistic environments. Large scale code studies cannot be easily controlled in a lab environment like their small scale code counterparts. Conversely, results from experiments that include small “toy” programs may not scale up. This problem is discussed in more detail in the *Problem* chapter.

This work builds theory through a field study with maintenance engineers understanding large-scale code in a production environment. The resulting model of program comprehension includes dynamic cognition behavior and information requirements. The field study involves programmers working on various maintenance tasks. Analysis of the observations extract cognition models and cognition processes for software maintenance.

Chapter Two contains a discussion of the problems addressed in this research. An overview of the literature demonstrates where this research fits into program comprehension research. Several deficiencies of existing research are addressed. A problem statement includes the major problem elements addressed and the scope of the problem.

Chapter Three provides background. It begins with definitions that will be used throughout the thesis. Common elements of typical code cognition models are presented including knowledge components, mental models, strategies, and characteristics of expert programmers. An *integrated code comprehension* model is described that incorporates two existing models found in the literature. This model is the basis for the research reported here. The final section deals with closely related research found in the literature.

Chapter Four covers the research design used for the study. The experimental design description includes a discussion of the subjects and the tasks they performed. An objectives section delineates a number of specific questions to be addressed in the analysis portion of the research. The chapter also covers a detailed description of the analysis methods used for the collected data.

Chapter Five describes the results and advances several hypotheses about the nature of program understanding for large scale, production code. We describe specific programmer actions taken during code understanding and dynamic comprehension processes which incorporate these actions. Generating hypotheses is an important aspect of understanding and we characterize hypothesis generation through the identification of hypothesis types and their resolution. We define an expertise indicator, the HK-Ratio, as the proportion of hypotheses generated and use of previously acquired knowledge. We show that maintenance engineers build mental representations of code at several levels of abstraction. We also illustrate that this process occurs by switching between all possible levels of abstraction at various times during program comprehension. Since programmers need specific



types of information when they understand code, we also analyzed for information needs and identified the tool capabilities required to meet those needs. We describe the minimum amount of information and expertise necessary to efficiently complete most maintenance tasks. Finally, we conclude with the most significant results. Chapter Five contains the aggregate results from our analysis. These results are based on subsets of subjects classified by type of task, level of expertise, and amount of prior experience with the software. Three appendices supply additional detail for the results section. Raw results from the analysis of individual subject data are presented in Appendix C. Detailed results from the analysis of aggregated individual subject results are presented in Appendix B.

Chapter Six proposes several follow-on experiments for validating our main conclusions. Our results appear to differ when based on task, expertise, and accumulated knowledge, however, our sample size was small and the differences we found may be due to individual differences. Therefore, in designing validation experiments, we suggest controlling task, expertise, and amount of accumulated experience with the code.

## Chapter 2

### THE PROBLEM

#### 2.1 Rationale

*Program Understanding* or *Code Cognition* is a central activity during software maintenance, evolution, and reuse. There are estimates that up to 50% [39] of the maintenance effort is spent in trying to understand code. A better grasp of how programmers understand code and what is most efficient and effective should lead to a variety of improvements: better tools, better maintenance guidelines and processes, and documentation that supports the cognitive process.

Maintenance encompasses many different tasks. Table 2.1, loosely based on Chapin's Software Maintenance Life-cycle [11], lists the most common. *Adaptive* maintenance occurs as a result of a change in the use or environment of an existing system and typically includes adding new code. *Perfective* maintenance involves adding functionality or *enhancements* to an existing system. *Corrective* maintenance consists of correcting errors in the product. Corrective maintenance usually occurs after program code is released for production and comprises the smallest portion of maintenance done on existing systems. *Reuse* involves using previously written code during development of new software. *Code Leverage* reuses code that is modified in some way to fit the problem at hand. We distinguish reuse from code leverage by restricting the term *reuse* to refer only to code modules that are reused *as-is*.

Each major maintenance task listed in column one of Table 2.1 consists of a variety of activities. Associated with these maintenance activities are specific objectives and each task may involve different methods of understanding code.

Experiments reported in the literature range from identifying the types of knowledge used during comprehension to the development of models of programming strategies that

Table 2.1: Maintenance Life-cycle Subtasks

<i>Maintenance Task</i>	<i>Activities</i>	<i>Comments</i>
Adaptive	Understand system	Understand existing system
	Define adaptation requirements	Requirements are extended from original requirements
	Develop prelim & detailed adaptation design	More design constraints due to existing system
	Code Changes	Merge adaptation code into existing code
	Debug	Focus on correct behavior of recently added code
	Regression tests	Develop tests for new code, run old tests to ensure other code not affected
Perfective	Understand system	Understand existing system
	Diagnosis & requirements definition for improvements	Identify exact nature of need for improvement e.g. performance improvements
	Develop prelim & detailed perfective design	Requirements are extended from original requirements
	Code changes/additions	Merge perfective code into existing code
	Debug	Focus on correct behavior of recently perfected code
	Regression tests	Develop tests for perfected code, run old tests to ensure other code not affected
Corrective	Understand system	Understand existing system
	Generate/Evaluate hypotheses concerning problem	Hypotheses about nature of problem generated using symbolic code execution, watching system behavior
	Repair code	Rewrite defective code or add omitted code
	Regression tests	Test for system stability after change
Reuse	Understand the problem, find solution based on close fit with predefined components	Reuse code AS IS, Design is open-ended, not initially constrained but defined by plugging some configuration of reusable components
	Obtain predefined components	Find reusable components
	Integrate predefined components	Similar to integrating new code developed by several engineers
Code Leverage	Understand problem, find solution based on predefined components	Design fixed or constrained by problem, design exists & close to desired solution, may use new code
	Reconfigure solution to increase likelihood of using predefined components	Find all possible solutions, use solution with highest leverageable code
	Obtain & modify predefined components	Find leverageable components
	Integrate modified components	Similar to integrating new code developed by several engineers

draw on this knowledge. Table 2.2 summarizes a representative sample of these experiments by type of maintenance task. Comparing this table to Table 2.1 demonstrates that not all maintenance tasks have been investigated for their cognitive processes. For example, no experiments have investigated reuse or code leveraging and with the exception of the last (number 18) row entry in Table 2.2, adaptive maintenance has not been investigated.

Tables 2.3 and 2.4 contain goals for each experiment listed in Table 2.2 (what was measured) and a bibliographic reference. In general, existing empirical studies of programming behavior fall into three broad areas:

- Expert versus novice behavior
- Cognitive elements of program development
- Cognitive elements of program maintenance

Whereas understanding is an important component for all three areas, this research concentrates on the last category: Cognitive elements of program maintenance. While Table 2.2 does not contain an exhaustive list, it is intended to show the variety of experimental procedures, tasks, participants, and programming languages that have been used to develop the current body of program understanding research.

In Table 2.2, the *Purpose* column lists the task focus for each experiment. *Understanding* is a maintenance task in and of itself when responsibility for maintaining code is first assigned. The person assuming the responsibility will want to understand (at least at a high level) the code before performing any maintenance. Other purposes are corrective, perfective, enhancements, and adaptations.

The *Code Size* column describes the code used in the understanding task in terms of size and structure. Most experiments have been done with code size less than 950 lines of code (LOC). 950 LOC is not representative of typical large-scale production code. This raises the issue of whether the results from these experiments using small-scale code will apply to large scale systems. For example, if the code is small enough, it is reasonable to expect an approach to understanding using a *systematic strategy* (code is read line by line). Koenemann & Robertson [27] (number 16 in Table 2.2) found that the programmers who used this systematic strategy were more successful at understanding code than those that used another approach. However, for systems of 50,000 LOC or more, the systematic approach is simply not feasible. The above experiment may not scale up because the program upon which it is based was only 636 lines of code. Similar questions can be asked about the results of any of the experiments using very small programs, i.e. will their results hold as the size of code increases?

The *Subjects/Expertise* column reports the level of expertise and the size of the participant sample. Novice participants are typically first year programming students. In-

intermediate subjects are also usually students who have had slightly more experience than novices, i.e. second or third year computer science students. Experts come from either a professional or computer science graduate student population. For large-scale production code, studying novice behavior is not appropriate, since presumably no company would hire a first year programming student to maintain a large system. A similar argument applies to intermediates. Professional programmers are the most highly desirable and if we are interested in knowing what experts do to make cognition more efficient, it makes the most sense to study them. We can use this understanding to infer guidelines for programmers with less expertise.

The *Language* column provides insight into the types of programming languages used for the experiments. The two most commonly used languages in these experiments are Pascal and Fortran, with Cobol the next most frequently used. It makes sense to include these languages in program comprehension studies. There are many very large legacy systems written in Fortran and Cobol that still need to be maintained. However, systems have been developed in languages such as C/C++ and their system and support environments, for example *Make*, *lint*, shell programs, *Motif*, *lex/yacc* etc. Thus, more studies are needed using more of these languages and support tools to see whether they affect program understanding.

The *Experimental Analysis &/or Design* column provides a brief description of the type of experiment used in each study. It describes the type of measurement used to capture the relationship between a specific (or set of) circumstance(s) and observed behavior. (A definition of the experimental design is given if the statement of the measurement procedure does not provide a suitable description.) For example, *protocol analysis* refers to the analysis of audio- and/or video-taped sessions of programmers working on a task.

The last column is a pointer to the reference which can be found in Tables 2.3 and 2.4. Tables 2.3 and 2.4 are continuations of Table 2.2 with the *No.* column of the first table corresponding to the number found in column one of the second set of tables.

Tables 2.3 and 2.4 describe the objective of each experiment and the associated literature reference. The objectives demonstrate that many experiments are focused on small aspects of maintenance activities. For example, experiments number 7 and 12 examine knowledge structure but not the application of that knowledge. Experiment number 3 is concerned only with the effect of mnemonic variable and procedure names on comprehen-

sion. Experiment number 18 defines an integrated model which describes how most of the components in the small-scale experiments might fit together for large-scale cognition needs.

Table 2.2: Comprehension Experiments

<i>Purpose</i>	<i>Code Size</i>	<i>Subjects/ Expertise</i>	<i>Language</i>	<i>Experimental Analysis&amp;/or Design</i>	<i>No.</i>
Understand	30 LOC (Modular)	24 Novice	BASIC	Verbatim Recall	(1)
	12-42 LOC (Modular)	22 Intermed. Students	Pascal	Cloze Test (Fill-in-blank) Protocol Analysis	(2)
	913 LOC (Modular)	10 Novice 8 Professional	C	Measure response time to hypothesis & location questions re: code	(3)
	12-42 LOC (Module)	7 Grad Students	Pascal	Cluster Analysis	(4)
	28-77 LOC (Modular)	30 Experts (Grad Students)	Pascal & Fortran	Frequency of segmented code fragments (major components). Match to predetermined plans	(5)
	136 LOC (Modular)	15 Experts Grad Students	Pascal	Recorded reading time & search patterns Protocol Analysis	(6)
	20 LOC (Module)	52 Novice 18 Intermed. 9 Grad Students	Fortran	Verbatim Recall	(7)
	15 LOC (Modular)	94 Novice 45 Grad Students	Pascal	Cloze Test (Fill-in-blank)	(8)
	250 LOC (Modular)	20+ Professional	Fortran	Protocol Analysis	(9)
	25 LOC (Module)	40 Novice 40 Intermed.	Pascal	Program Recall Function Description	(10)
Corrective	400 LOC (Modular)	8 Experts 8 Novices	Cobol	Protocol Analysis	(11)
	67 LOC (Modular)	38 Professional 42 Novice	Cobol	Program Recall	(12)
Perfective (Enhancement)	250 LOC (Several Modules)	4 Experts 2 Intermed.	Fortran	Protocol Analysis	(13)
	250 LOC (Modular)	10 Professional	Fortran	Protocol Analysis	(14)
Modifications	123-373 LOC (Object-oriented, Modular & All one function)	18 Intermed. (Students) 18 Professional	Pascal	Free Recall	(15)
	636 LOC (Modular)	12 Experts	Pascal	Time to complete Modification& Protocol Analysis	(16)
	15 LOC (Module)	80 Experts Professional	50% Cobol 50% Fortran	Free-recall & Response time to Comprehension Questions	(17)
	200 LOC (Modular)	40 Experts Professional	50% Cobol 50% Fortran	Comprehension Questions & Protocol Analysis	(17)
Understand, Corrective, Adaptation, & Perfective	50-80,000+ LOC (Module-Sys)	11 Experts Professional	Procedural (e.g C, C- shell,etc)	Protocol Analysis	(18)

Table 2.3: Experimental Purposes & References

No.	Experiment Designed to Measure:	Reference
(1)	If plans are closely tied to the nature of design experience	Simon P. Davies, <i>The nature and development of programming plans</i> , In: International Journal of Man-Machine Studies, 32(1990), pp. 461 - 481.
(2)	To Identify understanding process strategies in plan vs. unplan-like situations	Francoise Detienne & Elliot Soloway, <i>An empirically-derived control structure for the process of program understanding</i> , In: International Journal of Man-Machine Studies, 33(1990), pp. 323-342.
(3)	Do beacons help experts read & understand large programs	Edward M. Gellenbeck & Curtis R. Cook, <i>Does Signaling Help Professional Programmers Read and Understand Computer Programs?</i> , Empirical Studies of Programmers: 4th Workshop, 12/91, New Brunswick, NJ, pp. 82-98.
(4)	If programmers have and use plans	Robert S. Rist, <i>Plans in Programming: Definition, Demonstration, and Development</i> , In: Empirical Studies of Programmers: 1st Workshop, 1986, Washington, D.C., pp. 28-47.
(5)	Examine content of plan-based representations	Scott P. Robertson & Chiung-Chen Yu, <i>Common cognitive representations of program code across tasks and languages</i> , In: International Journal of Man-Machine Studies, 33(1990), pp.343-360.
(6)	If code comprehension is a problem solving task.	S. P. Robertson, E. F. Davis, K. O., & D. Fitz-Randolf, <i>Program Comprehension Beyond the Line</i> In: INTERACT '90, D. Diaper et al. (Eds.), Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 959 - 970. pp. 959 - 970.
(7)	If experts encode programs as chunks of information	Ben Shneiderman, <i>Exploratory Experiments in Programmer Behavior</i> , In: International Journal of Computer and Information Sciences, Vol 5, No. 2, 1976, pp. 123 - 143.
(8)	If programmers encode programs in chunks of information	Elliot Soloway & Kate Ehrlich, <i>Empirical Studies of Programming Knowledge</i> , In: IEEE Transactions on Software Engineering, September 1984, Vol. SE-10, No. 5, pp. 595-609.
(9)	To find the types of representations that help programmers when making changes to programs.	Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman & Robin Lampert, <i>Designing Documentation To Compensate For Delocalized Plans</i> , In: Communications of The ACM, Vol. 31, No. 11, November 1988, pp. 1259-1267.
(10)	If beacons help make program comprehension easier.	Susan Wiedenbeck, <i>The initial stage of program comprehension</i> , In: International Journal of Man-Machine Studies, 35(1991), pp. 519-540.
(11)	Characterize expert debugging strategies	Iris Vessey, <i>Expertise in debugging computer programs: A process analysis</i> , In: International Journal of Man-Machine Studies, (1985)23, pp.459-494.
(12)	If performance of programmers increase when knowledge structures match program structure plans	Iris Vessey, <i>On matching programmers' chunks with program structures: An empirical investigation</i> , In: International Journal of Man-Machine Studies, (1987)27, pp.65-89.
(13)	If programmers understand code in Bottom-up and Top-down fashion	Stanley Letovsky, <i>Cognitive Processes in Program Comprehension</i> , In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 58 - 79.
(14)	To find a relationship between understanding strategies (systematic & opportunistic) & modifying a program successfully.	David C. Littman, Jeannine Pinto, Stanley Letovsky, & Elliot Soloway, <i>Mental Models and Software Maintenance</i> , In: Empirical Studies of Programmers, Eds. Soloway & Iyengar, ©1986, Ablex Publishing Corporation, pp. 80 - 98.



Table 2.4: Experimental Purposes & References (continued)

No.	Experiment Designed to Measure:	Reference
(15)	Measure the effects of program structure, content, and ease of modification on mental models	Robert W. Holt, Deborah A. Boehm-Davis, & Alan C. Schultz, <i>Mental Representations of Programs for Student and Professional Programmers</i> , In: Empirical Studies of Programmers: Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 33 - 46.
(16)	If programmers use systematic & opportunistic strategies in program understanding	Jurgen Koenemann & Scott P. Robertson, <i>Expert Problem Solving Strategies for Program Comprehension</i> , In: Proceedings of CHI'91, March 1991, pp. 125-130.
(17)	If program control-flow (Program Model) is understood before functional view (Situation Model)	Nancy Pennington, <i>Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs</i> , In: Cognitive Psychology, 19(1987), pp.295-341.
(18)	If programmers understand code according to the integrated model	A. von Mayrhauser & A. Vans, <i>From Code Understanding Needs to Reverse Engineering Tool Capabilities</i> , In: Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE93), Singapore, July 1993, pp. 230 - 239.

When looked at collectively, several opportunities for investigation emerge.

#### 1. Maintenance Tasks

First, these experiments fail to provide a clear picture of comprehension processes based on specialized maintenance tasks. Several experiments described in the literature characterize general understanding. However, the objectives in these are typically focused on cognitive aspects such as differences between novice and expert programmers or whether the naming conventions and program structure aid in comprehension. Less than 50% of these experiments are designed to investigate particular maintenance tasks like debugging or program modification. While models of the general understanding process play a very important part in furthering insight into complete understanding of an entire piece of code, they may not be entirely correct for all maintenance tasks. For example, existing literature on general understanding reports that the most complete understanding occurs when code is systematically understood [27]. Any systematic approach to understanding requires that *all* code is understood in detail. For specific maintenance tasks like “find the interface error in X”, this does not seem feasible for large-size software. Instead, it would be preferable to identify specialized cognition processes that, because they have more focus, are more efficient.

## 2. *Static versus Dynamic Behavior*

The literature is also mainly focused on the *static properties* of programming skills [15]. For example, stable knowledge consisting of common program components such as searching and sorting algorithms has been studied, but experiments do not investigate the actual use and application of this knowledge. Studies of *static knowledge* are much more prevalent in the literature than studies of dynamic strategies which include the use and application of static knowledge.

## 3. *Scalability of experiments*

Another major issue is that the results of these experiments may not scale up for production code. For example, Vessey's study of expert programmers' knowledge [52] and Pennington's study of mental representations of code [36] used programs of lengths varying between 67 LOC and 200 LOC. These studies are appropriate for answering questions about understanding small program segments or very small programs. However, nothing can be concluded about the interactions of these types of isolated components for understanding nor whether these activities will play an important role in understanding large programs.

## 4. *Theory Building*

At the root of the previous three issues lies a common problem: many experiments are designed to measure specific conditions (i.e. do programmers use plans <sup>1</sup>) but the experimental conjectures (programmers use plans when understanding code they have never seen before) are not based on a well-defined theory of program comprehension. Sheil [43] concludes that "Our primary need at the moment is for a theory of programming skill that can provide both general guidance for system designers and specific guidance to psychologists selecting topics for detailed studies. The experimental investigation of such factors as the style of conditional notation is premature without some theory which gives some account of why they might be significant factors in programmer behavior." Although this paper was written in 1981, in the

---

<sup>1</sup>Plans are elements of knowledge that are stereotypic structures of common programming concepts, e.g. bubble sort

past 15 years very few theories concerning program comprehension have been advanced. No theories regarding large scale program comprehension for specialized maintenance tasks have been developed.

## 2.2 The Problem Statement

The purpose of this research is to construct a theory of large scale program understanding by developing a model of comprehension during maintenance. To accomplish this, the research focuses on the four areas already discussed:

- Concentration on the understanding needs and processes of specific maintenance tasks
- Investigation of both Static and Dynamic comprehension behavior
- Providing a comprehension theory for large-scale, production code
- Building a consistent, comprehensive theory

The theory is developed through a study of programmers performing various maintenance tasks on production code. The main focus is on building a model of large-scale program comprehension that incorporates both static and dynamic elements. The starting point is the *Integrated Comprehension Model* (described in [62, 63, 65]) which is composed of three other models [36, 49]. Both static and dynamic comprehension behaviors are investigated under the assumptions of the Integrated Comprehension Model. Analysis of comprehension behaviors lead to the identification of strategies associated with specific maintenance tasks. Thus, the issues of scalability, static and dynamic behavior, together with models of program comprehension based on actual maintenance tasks are considered.

## 2.3 Elements of the Problem

Several aspects of the problem statement are examined in this research. This section presents them as questions to focus our study:

- What happens when an engineer assumes responsibility for a piece of code that is completely new to her/him and tries to understand it? What questions do they ask and what information is most helpful during understanding? What types of hypotheses do they generate and under what conditions are these hypotheses confirmed or rejected?
- What effect does expertise and amount of prior experience with the code have on the types of actions taken, hypotheses generation, and how mental representations are constructed?
- Can we identify a common understanding process? Are there different understanding processes based on maintenance task, expertise, or amount of prior experience with the code?
- Can we classify information needs by type of maintenance task? Since programmers understand code for a specific reason, usually related to maintenance task, the types of information used may differ between tasks.

#### 2.4 Scope of the Study

The scope of our research is to contribute to a theory of large-scale program understanding. The starting point was the *Integrated Code Comprehension Model* we constructed, based on existing research. The main contribution of this research is to develop a theory of program comprehension based on this model. The theory includes several levels of detail encompassing static and dynamic aspects of code understanding including processes, programmer actions, and information needs. Think-aloud reports from 11 subjects constitute the data set. Although this data set is small, the subjects represent a diverse population of maintenance tasks and other variables such as expertise and accumulated knowledge about the specific task. Since it is impossible to control all variables in a study such as ours, we emphasize its purpose as building a theory through a careful analysis and evaluation of the data. Further controlled experimentation will be required to properly validate the theory built based on these observations.

## Chapter 3

### BACKGROUND

#### 3.1 Overview

This chapter describes the major components of code comprehension and presents a review and evaluation of related code comprehension models. The chapter concludes by demonstrating how this research adds to the body of knowledge in code comprehension.

#### 3.2 Common Elements of Cognition Models

Program comprehension is a series of processes that use *existing knowledge* to acquire *new knowledge* that ultimately meets the objectives of the code cognition task at hand. During program comprehension programmers reference both existing and newly acquired knowledge to build a *mental model* of how the software works. A mental model is defined as a programmer's internal representation of the code. How programmers go about understanding depends on *strategies*. A strategy guides the sequence of actions while following a plan of action. For example, if the goal is to understand a block of code, the strategy may be to use a systematic approach by reading and understanding each line of code. While cognition strategies vary, they share the process of formulating *hypotheses* and then resolving, revising, or abandoning them. An hypothesis is a theory or assumption about some aspect of the code.

##### 3.2.1 Knowledge

Programmers must have two types of knowledge, general knowledge that is independent of the specific software application they are trying to understand, and software specific knowledge that represents their current level of understanding of the software application. During the understanding process they acquire additional software specific

knowledge, but may also need more general knowledge (e. g. how a round-robin algorithm works). Existing knowledge includes knowledge of programming languages and computing environments, programming principles, choices specific to the application domain architecture, algorithms, and possible solution approaches. In cases where the programmer has worked with the code before and knows something about it, existing knowledge includes any (partial) mental model of the software.

New knowledge is primarily knowledge about the software product. It is acquired throughout the code understanding process as the mental model is built. This knowledge relates to functionality, software architecture, how algorithms and objects are implemented, control, and data flow, etc. New knowledge spans a variety of levels of abstraction from “this is an operating system” to “variable q is incremented in this loop”.

The understanding process matches existing knowledge with newly acquired knowledge about the software until there are enough matches to satisfy the programmer that she understands the code. The set of matches is the mental model. It can be complete or incomplete.

### 3.2.2 Mental Model

The mental model is the programmer’s current internal (working) representation of the software under consideration. It is constructed using a hierarchy of various knowledge structures including *text structures*, *chunks*, and *plans*. Top level plans specialize into other plans or chunks. Each chunk in turn, represents a higher level abstraction of other chunks or text structures.

#### 3.2.2.1 Text Structure

Text structure knowledge includes the program text and its structure. Text-structure knowledge [26] is acquired through experience and is subsequently stored in long-term memory. Pennington [36] uses text-structure knowledge to explain control-flow knowledge for program understanding. Structured programming units form text structure and are the general organizing structure in her comprehension model.

Examples of text structure knowledge units are: *Control-Primes* – Iteration (loop constructs), sequence, and conditional constructs (eg. if-then-else); *Variable definitions*;

*Module calling hierarchy*; and *Module parameter definitions*. This *Micro-structure* of the program text consists of the actual program statements and their relationships. For example, the statement **BEGIN** signifies the start of a block of code, while a subsequent **IF** indicates a conditional control structure for some purpose. The relationship between these two propositions is that the **IF** is part of the block initiated by the **BEGIN**.

### 3.2.2.2 Chunks

Chunks are knowledge structures consisting of various levels of abstractions of text structures. Chunks of text-structure, designated as *Macro-structures*, are identified by a label and correspond to control-flow organization of the program text [36]. For example, the micro-structure for a sort consists of all its statements. The macro-structure is an abstraction of the block of code and consists only of the label *sort*. Lower level chunks can form higher level chunks. Higher level chunks consist of several labels and the control-flow relationships between them.

### 3.2.2.3 Plans

Plans are elements of knowledge that support the development and validation of expectations, interpretations, inferencing, and help programmers stay focused during the program understanding task. These plans also include causal knowledge which is the actual flow of information and relationships between parts of programs. Plans are schemas or *frames* which are knowledge structures with two parts: slot-types (or templates) and slot fillers. Slot-types describe generic objects while slot fillers are customizations that fit a particular feature. Data structures like lists or trees are examples of slot-types and specific program fragments are examples of slot-fillers. These structures are linked by either a *Kind-of* or an *Is-A* relationship.

*Programming plans* can be low level, intermediate level, or high level programming concepts. Iteration and conditional code segments are low level concepts that can be plans or components of plans that exist as a chunk of text structure. Searching, sorting, and summing algorithms as well as data-structure knowledge including arrays, linked-lists, trees, and stacks are intermediate level. Intermediate level plans can be aggregated into

functional units comprising high level plans. Programming plan knowledge includes roles for data objects, operations, tests, other plans, and constraints on what can fill the roles.

*Domain plans* incorporate all knowledge about the problem area except for code and low-level algorithms. Domain plans apply to objects in the real world. For example, useful plans when developing a software tool for designing automobiles include schemas related to the function and appearance of a generic car. Slots for problem domain objects such as steering wheels, engines, doors, and tires are necessary components of an appropriate plan. For program understanding, these plans are crucial for understanding program functionality. Control-flow plans alone are not enough to understand aspects such as causal relationships among variables and functions. Domain plans are also concerned with the environment surrounding the software application, the domain specific architecture and solution alternatives.

In short, the mental model is one plan composed of several sub-plans representing different levels of abstraction. Each plan represents software specific or software independent information with slots and fillers for other plans or chunks of text structure.

### 3.2.3 Hypotheses

Letovsky [29] refers to hypotheses as *conjectures* and defines them as comprehension activities (actions) that take on the order of seconds or minutes to occur. Letovsky identified three major types of hypotheses: *Why* conjectures which hypothesize the purpose of some function or design choice, *How* conjectures hypothesize about the method for accomplishing a program goal, and *What* conjectures which hypothesize about what something is, for example a variable or function. Additionally, there are degrees of certainty associated with a conjecture and these vary from uncertain guesses to almost certain conclusions.

Brooks [9] theorizes that hypotheses are the only drivers of cognition. This theory states that understanding is complete when the mental model consists entirely of a complete hierarchy of hypotheses. At the top of this hierarchy is the *primary hypothesis* which is a high-level description of the program structure. It is necessarily global and non-specific. Once the primary hypothesis is generated, subsidiary hypotheses that support the primary hypothesis are generated. This process is continued until the mental model



is built. Brooks also considers three reasons hypotheses sometimes fail: code to verify an hypothesis cannot be found; confusion due to a single piece of code that satisfies different hypotheses; and code that cannot be explained.

### 3.2.4 Strategies

A strategy guides the sequence of actions while following a plan to reach a particular goal. For example, if the goal is to understand a block of code, the strategy may be to approach the comprehension task *systematically* by reading and understanding every single line of code while building a mental representation at higher and higher levels of abstraction. Programmers who use an *opportunistic strategy* study code in a more haphazard fashion. Littman et al [30] found that people who used a systematic approach to comprehension were more successful at modifying code (once they understood it) than programmers who took the opportunistic approach. On the other hand, for large programs systematic understanding may not be possible.

Strategies also differ in matching programming plans to code. *Shallow reasoning* [48, 49] does so without in-depth analysis. Many experts do this when they recognize familiar plans. *Deep reasoning* [48, 49] looks for causal relationships among procedures or objects and performs detailed analyses.

Strategies guide understanding mechanisms that produce information. Two such mechanisms are *chunking* and *cross-referencing* [37]. *Chunking* creates new higher level abstractions from *chunks* of lower level structures. As groups of structures are recognized, labels replace the detail of the lower-level chunks. In this way, lower level structures can be chunked into larger structures at higher levels of abstraction. For example, a piece of code may represent a linked-list definition as pointers and data. In an operating system definition this may be abstracted as a “ready-queue”. The section of code that takes a job from the ready queue, puts it into the running state, monitors elapsed time, and removes the job after the time quantum has expired may be abstracted with the label “round-robin scheduler”. The fragments of code for the queue, the timer, and the scheduling are micro-structures. Continued abstraction of round-robin scheduler, dead-lock resolution, interprocess communication, process creation/deletion, and process synchronization even-

tually leads to the higher level structure definition: “process management of the operating system”.

*Cross-referencing* relates different levels of abstraction, e. g. a control-flow view and a functional view by mapping program parts to functional descriptions. For instance, I can make a statement about functionality once I know that a segment of code performs process management and know why the code exists. Cross-referencing is thus an integral part of building a complete mental representation across all levels of abstraction.

If we look at code cognition as a process that formulates hypotheses and then checks whether they are supported or refuted and revises them where necessary, then hypotheses can be seen as programmer-defined goals. Programmers are trying to match these goals. Goals exist at all levels of abstraction, like plans and schemas. The essence of an effective and efficient strategy is to keep number of goals manageable while increasing understanding incrementally.

### **3.2.5 Facilitating Knowledge Acquisition**

#### **3.2.5.1 Beacons**

Beacons are cues that index into knowledge. Making them visible facilitates code understanding. Beacons can be text or a component of other knowledge. For example, a swap statement inside a loop or a procedure called *Sort* can act as a beacon for a sorting function. Wiedenbeck [67] used short Pascal programs in a recall experiment (i.e. programmers had to recall code they had previously seen, verbatim) designed to study whether programmers actually use beacons during program comprehension activities. Experienced programmers were able to recall beacon lines much better than novices. Beacons are useful for gaining a high level understanding in processes such as top-down comprehension.

Gellenbeck & Cook [22] investigated the role of mnemonic procedure and variable names as beacons in understanding code. While the study confirmed the usefulness of beacons in general, no useful conclusions regarding the strength of variable names and procedure names could be shown.

The *Book Paradigm* developed by Oman and Curtis [33, 34] is a tool that highlights beacons by using typographic arrangements of source code. Experimental results indicate that programmers can benefit substantially using source code that is reformatted with the book paradigm technique.

### 3.2.5.2 Rules of discourse

Rules of discourse are conventions in programming, similar to dialogue rules in conversation. Examples are coding standards, common forms of algorithm implementations, expected use of data structures, mnemonic naming, etc. Rules of discourse set expectations of programmers. Programmers retrieve programming plans from long term memory using these expectations. Soloway and Ehrlich [48] showed by a recall experiment that rules of discourse had a significant effect on the ability of expert programmers to comprehend code. The experiment required programmers to understand one program that was written using *plan-like* code and a second program that used *unplan-like* code. Plan-like code is defined as code fragments that match expert programming plans. They were able to show that the programmers performed significantly better on the plan-like code than on the unplan-like code. The performance of these experts dropped to that of novices when attempting to understand the unplan-like code since they were unable to match this code to any programming plans stored in long-term memory. In practice this means that unconventional algorithms and programming styles are much harder to understand, even for experts.

### 3.2.6 Expert Characteristics

The level of expertise in a given domain greatly affects the efficiency and the success of a programmer during program understanding. Experts tend to show the following characteristics:

- Experts organize knowledge structures by functional characteristics of the domain in which they are experts. Knowledge possessed by novices is typically organized by surface features of the problem. For instance, novices may have knowledge about a particular program organized according to the program syntax. An example of a functional category is algorithms. Experts may organize knowledge about programs

in terms of the algorithms applied rather than the syntax used to implement the program, [25].

- Experts have efficiently organized specialized schemas developed through experience. A high-level design study conducted by Guindon, [25], indicated that experts not only used general problem solving strategies such as divide-and-conquer, but also more specialized design schemas. These schemas differed in granularity and seemed to be abstracted from previously designed software systems. The schemas had comparable structures, but different problem domains.
- Specialized schemas contribute to efficient problem decomposition and comprehension, [25]. For problems that match specialized schemas, top-down comprehension becomes feasible.
- Vessey [53], conducted several debugging experiments and found that experts are flexible in approaches to problem comprehension. In addition, experts are not constrained by generated hypotheses. They are able to let go of questionable hypotheses and assumptions more easily. Experts tend to generate a breadth-first view of the program and then refine hypotheses as more information becomes available.

### 3.3 Models

This section contains a description of five models of program comprehension that are directly related to the research proposed. A discussion of the similarities between these five lead into an introduction of the Integrated Code Comprehension Model.

#### 3.3.1 Letovsky Model

Letovsky's comprehension model, [29], has three main components – a knowledge base, a mental model (internal representation), and an *assimilation process*. This model is a very high level cognitive model of program understanding. The knowledge base consists of programming expertise, problem domain knowledge, rules of discourse, plans (similar to Pennington's text-structure knowledge and plan knowledge), and goals.

The mental model consists of three layers – a specification, an implementation, and an annotation layer. The specification layer contains a complete characterization of the program goals. This is also the highest level of abstraction of the program. The implementation layer contains the lowest level abstraction with data structures and functions as entities. The annotation layer ties each goal in the specification layer to its realization in the implementation layer.

The assimilation process can occur in either a top-down or bottom-up fashion. It is opportunistic in that the programmer proceeds in a way she feels yields the highest return in the form of knowledge gain. Understanding proceeds by matching code, documents, etc. to elements from the knowledge base. The sole purpose of this matching process is to contribute to one of the three layers constructed in the mental representation. During the understanding process there may be problems connecting specification and implementation layers when it is not immediately known how to implement some function. The *dangling purpose unit* models these unresolved links between the specification and implementation layers. Figure 3.1 graphically represents Letovsky's model.

## Model of Comprehension — Letovsky

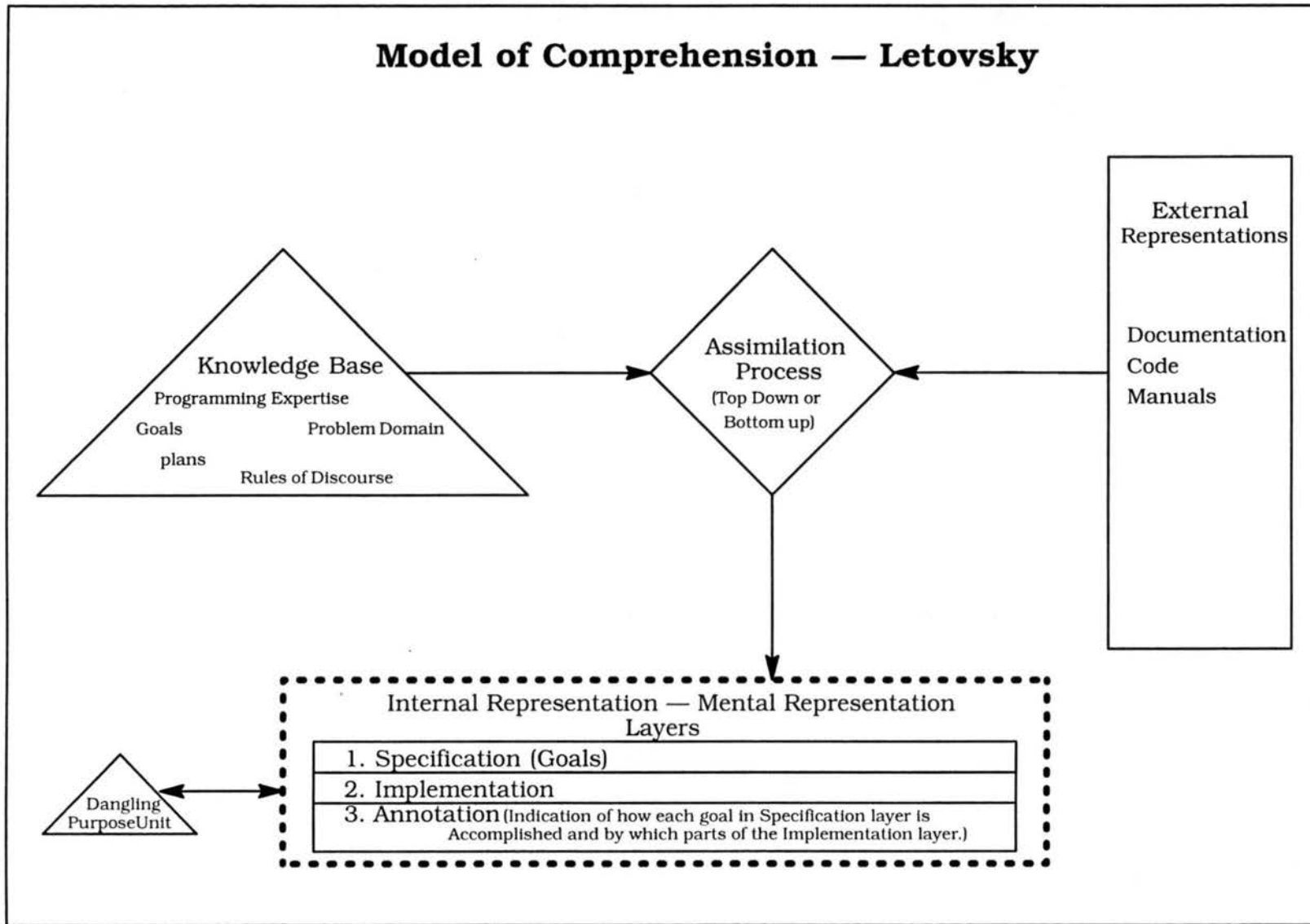


Figure 3.1: Letovsky — Comprehension Model

### 3.3.2 Shneiderman Model

The Shneiderman comprehension model is shown in Figure 3.2 [45]. Program comprehension involves recoding the program in short-term memory via a chunking process into an *internal semantic representation* using working memory. These internal semantics consist of different levels of abstraction of the program. At the top are high-level concepts like program goals. At the lowest levels are details such as the algorithms used to achieve program goals.

Long-term memory helps during internal semantics construction. Long-term memory is a knowledge base with semantic and syntactic knowledge. Syntactic knowledge is programming language dependent while semantic knowledge consists of general programming knowledge independent of any specific programming language. Like working memory, semantic knowledge in long-term memory is multi-leveled and incorporates high-level concepts and low-level details. Design works forward from the problem statement to the program while program understanding starts with the program and works to the problem statement.

# Model of Comprehension — Shneiderman

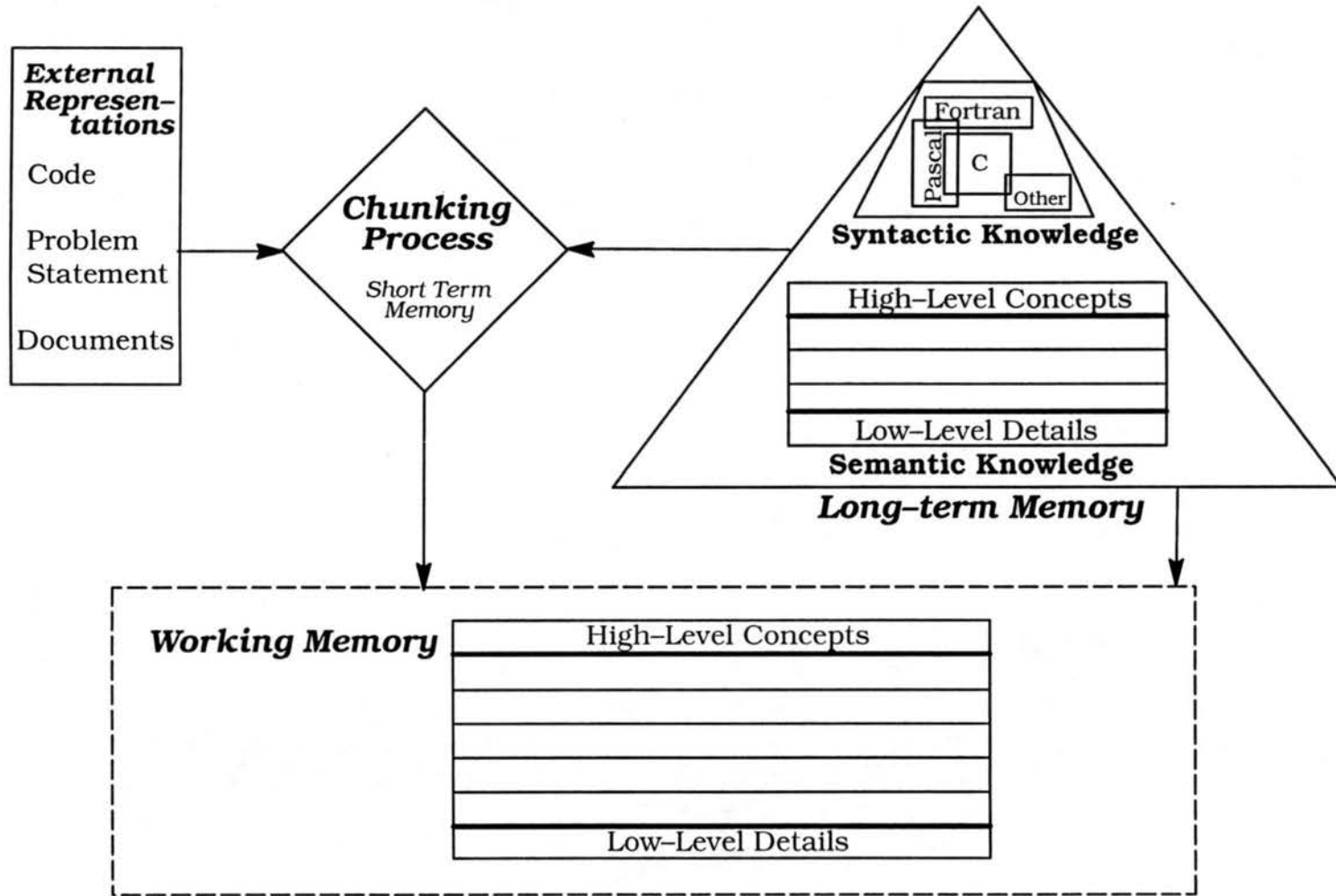


Figure 3.2: Shneiderman – Comprehension Model



### 3.3.3 Brooks Model

Brooks, [9], defines program comprehension as the reconstruction of the domain knowledge used by the initial developer. Domain knowledge is knowledge about specific domains such as compilers or operating systems. In this theory, understanding proceeds by recreating the mappings from the problem domain through several intermediate domains into the programming domain. The problem domain or *application domain* consists of problems in the real world.

An example of a problem in the application domain might be the maintaining of appropriate levels of inventory in order to keep back-orders to a minimum and at the same time minimizing exposure to loss due to obsolete inventory. The objects are inventories whose levels must be closely monitored to meet the constraints of the problem. These are physical entities that have properties such as size, cost, and quantity. In order to construct a program to solve this problem, these objects and their properties must be encoded for use by a computer. Once the physical objects are characterized, intermediate knowledge domains are required. The inventory can be assigned part numbers. Perhaps cost is determined not only by actual cost but also overhead like storage. We need knowledge of accounting practices to recognize the appropriate overhead calculations. Once the equations are identified, we need knowledge of program syntax to implement the equations in a programming language. This example used at least four different knowledge domains to reach the programming domain: inventories, accounting, mathematics, and programming languages.

Knowledge within each domain consists of details about the objects in the domain, the set of operations allowed on the objects, and the order in which the operations are allowed. There is also inter-domain knowledge that describes the relationships between objects in different, but closely related domains such as operating systems in general and UNIX in particular.

The mental model is built through a top-down process that successively refines hypotheses and auxiliary hypotheses. Hypotheses pertain to specific domains or connections between knowledge domains. For instance, an hypothesis may state that a particular equation (math domain) expresses cost (accounting domain). Hypotheses can be generated through the recognition of beacons. For example, a procedure name *FCFS* may

generate the hypothesis that a first-come-first-serve algorithm is used for process scheduling. Hypothesis generation drives domain knowledge retrieval. Hypotheses are iteratively refined, passing through several knowledge domains, until they can be matched to specific code in the program or some related document.

Figure 3.3 depicts this model. Knowledge, shown as triangles, can be used directly for hypothesis generation in the mental model or it can be matched (mapped) from one domain into another. Another cognitive process verifies that internal representations reflect knowledge contained in external representations such as code, design documents, or requirements specifications. Beacons are the main vehicle for this verification and can be used to look at either the internal or external representations for expected information. Verification is also hypothesis driven in that once an hypothesis is generated the external (internal) representations can be searched to support the hypothesis.

## Model of Comprehension — Brooks

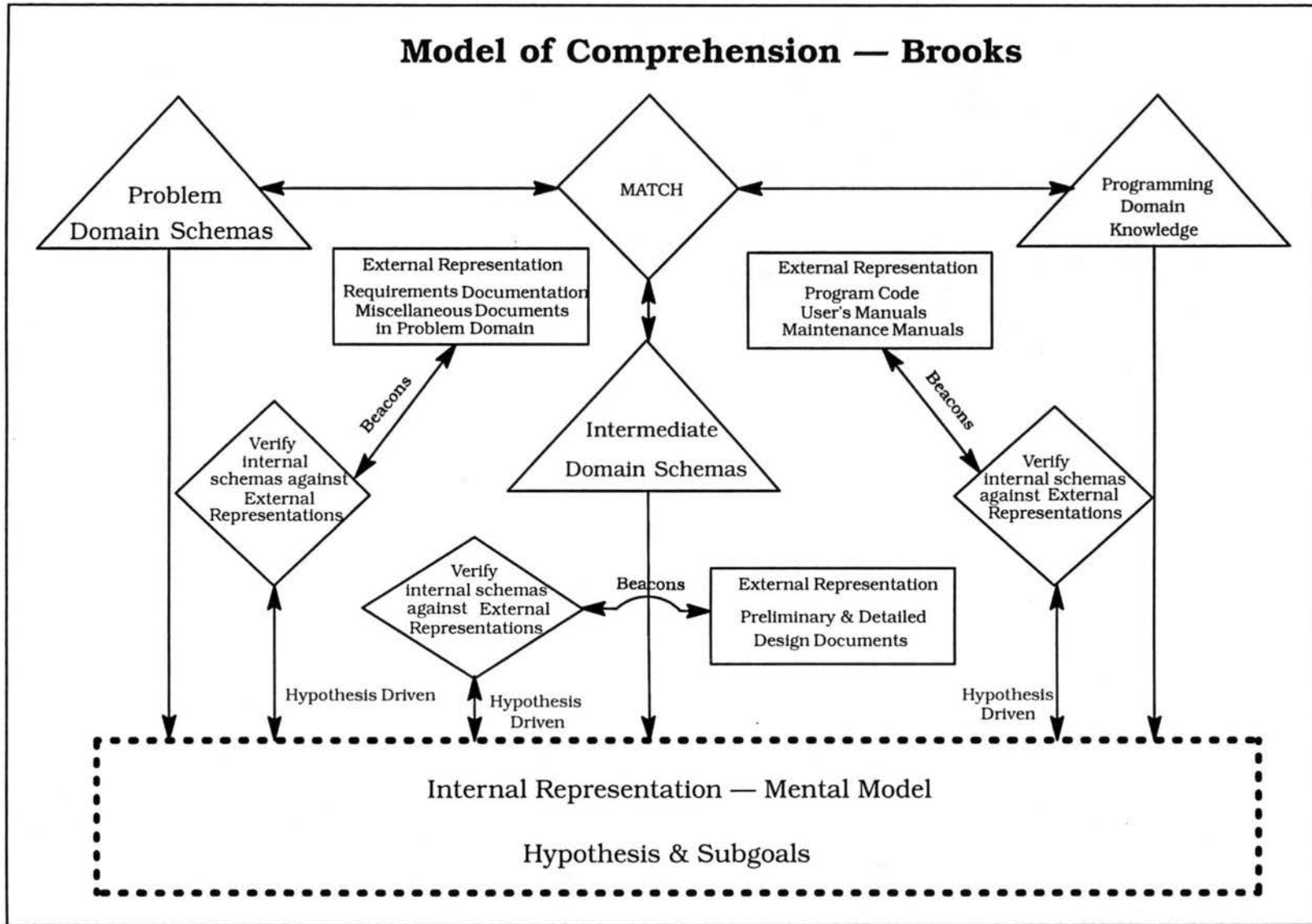


Figure 3.3: Brooks — Comprehension Model

### 3.3.4 Top-Down – Soloway & Ehrlich

*Top Down* program understanding [48, 49] typically happens when the code or type of code is familiar. Suppose an expert whose specialty is operating systems, is asked to maintain an operating system she has never before seen. As an expert, she can immediately decompose the new system into elements she knows must be implemented in the code: a process manager, a file manager, an I/O manager, and a memory manager. Each of these can be decomposed e.g, process management includes interprocess communication and process scheduling. Process scheduling can be implemented through one of several scheduling algorithms: e.g. round robin, shortest job first, or priority scheduling. The programmer may continue in this top down fashion until she recognizes a block of code, in this case, the precise process scheduling algorithm. Notice that it is not necessary to re-learn this algorithm line by line. Instead, the engineer need only recognize that the appropriate code exists. Theoretically, new code could be understood entirely in a top down manner if the programmer had already mastered similar code and the new code was structured in exactly the same way.

This model uses three types of plans: *Strategic*, *Tactical*, and *Implementation* plans. *Strategic Plans* specify a global plan used in a program or an algorithm and specify actions that are language independent. These are the highest-level plans available during comprehension. An example of a strategic plan is the process state model for operating systems. These plans say nothing about the actual constructs used for implementation and contain no lower level detail. Strategic plans can be further decomposed into language independent tactical plans.

*Tactical plans* are local plans for solving a problem. These plans contain language independent specifications of algorithms. For an operating system, tactical plans might include cpu scheduling algorithms for the process state model, such as FCFS (First-Come First-Served), Shortest-Job-First, Priority scheduling, or Round-Robin. These knowledge structures may include abstract data-structures, such as queues to keep track of which process to schedule next. The tactical plans composed of these algorithm descriptions are linked to the *process state model* strategic plan. Tactical plans cannot be used directly for understanding specific code since they are not tied to particular languages. Tactical

plans can include the abstract data types or objects. For instance, a queue may be used in the FCFS algorithm.

*Implementation plans* are language dependent and are used to implement tactical plans. These plans contain actual code fragments acquired through experience. A First-Come First-Served function written in C is an example of an implementation plan. A queue for FCFS can be implemented as a linked list or array structure. These represent two different implementation plans for the same tactical plan.

A mental model is constructed during top down comprehension and consists of a hierarchy of goals and plans. Rules of discourse and beacons facilitate decomposition of goals into plans and plans into lower-level plans. Typically, a shallow reasoning strategy is used to build the connections between the hierarchical components.

Figure 3.4 shows the model's three major components: 1) The triangles represent knowledge (programming plans or rules of discourse). 2) The diamond represents the understanding process which is defined as a matching of documents to programming plans. 3) The rectangles illustrate internal or external representations of the program. Understanding matches external representations to programming plans using rules of discourse for help in selecting plans (by setting expectations). Once a match is complete, the internal representation is updated to reflect the newly acquired knowledge. These updated mental representations are subsequently stored as new plans.

Comprehension begins with a high-level goal and proceeds with the generation of detailed sub-goals necessary to achieve the higher level goals. The programmer draws on previously stored knowledge (plans) and programming rules of discourse in an attempt to satisfy the goals. Program documentation and code serve as the tools for invocation of implementation, strategic, or tactical plans, depending on the focus of the current mental representation. In addition to building the current mental representation, top down comprehension also facilitates the building of new programming plans which are in turn stored in long term memory for future use.

# Model of Comprehension — Soloway & Ehrlich

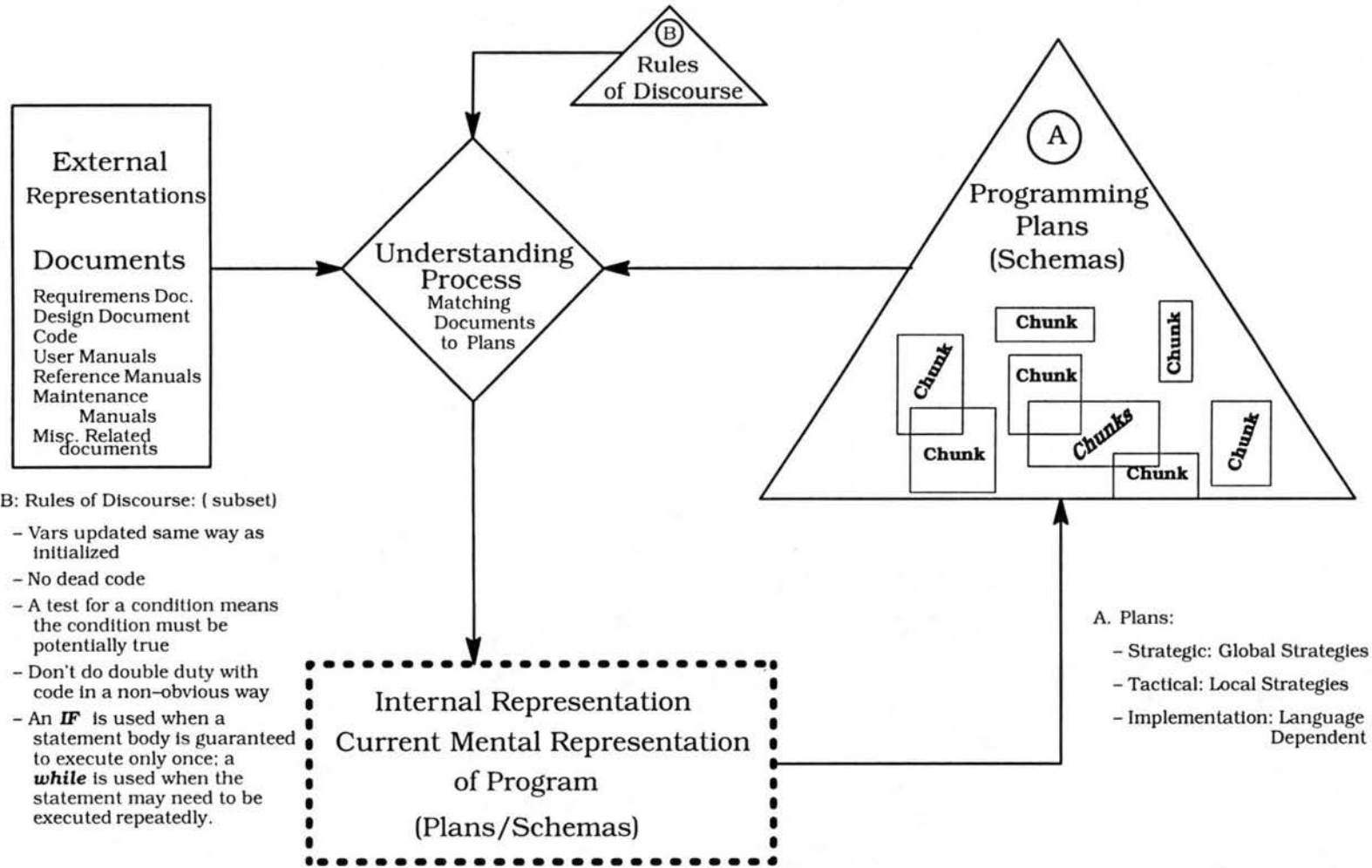


Figure 3.4: Soloway & Ehrlich – Comprehension Model

### 3.3.5 Pennington's Model – Bottom-Up Comprehension

Pennington's [36, 37] Bottom-Up Comprehension Model consists of two models: a Program Model which builds a mental representation of the actual code and a Situation Model which is an abstraction of the Program Model.

#### 3.3.5.1 The Program Model

When code to be understood is completely new to the programmer Pennington [36, 37], found that the first mental representation programmers build is a control flow abstraction of the program called the *program model*. This representation is built from the *bottom up* using *beacons* to identify elementary blocks of code (*control primes*) in the program. Pennington uses *text structure* and *programming plan knowledge* to explain the development of a program model. For people trying to understand programs, this text-structure knowledge is comprised of the control primes used to build the program model. Programming plan knowledge, consisting of more involved programming concepts, is used to exploit existing knowledge during the understanding task and to infer new plans for storage in long-term memory. Examples of plan knowledge structures from the Operating Systems domain are memory management page replacement algorithms including LRU (Least Recently Used) and NRU (Not Recently Used). Data structure knowledge may contain the implementation of a FIFO queue.

The mental representation is a current internal (working) description of program text and represents understanding. Two different representations are developed during comprehension – a *Program Model* and a *Situation Model*. The program model may be developed before the situation model. Text-structure and programming plan knowledge play a critical role in the development of the program model.

The program model is created by *chunking micro-structures* into *macro-structures* and by cross-referencing. Cross referencing allows modifications to the program model using connections into (from the situation model) and out of (to the situation model) the program model representation.

Figure 3.5 is a graphical representation of Pennington's model. The right half illustrates the process of program model building while the left half describes situation model construction. Text-structure knowledge and any external representations (code, design

documents, etc.) are inputs to the comprehension process. Beacons can influence invocation of a particular schema (e.g. a swap operation causes the programmer to recall sorting functions). Code statements and the interrelationships among them are organized into a micro-structure. Micro-structures are chunked into macro-structures. These chunks are stored in long-term memory and subsequently used in the comprehension process to build even larger chunks. Once a control-flow mental representation exists, the program model is established.

### 3.3.5.2 The Situation Model

Once the program model representation exists, Pennington, [36], showed that a *situation model* is developed. This representation, also built from the bottom up, uses the program model to create a data-flow/functional abstraction. Knowledge of real-world domains is required. For the operating systems example, this knowledge includes facts about generic operating system structure and functionality. Construction of the situation model is complete once the program goal is reached.

*Domain Plan Knowledge* is used to derive a mental representation of the code in terms of real-world objects and organized as a functional hierarchy in the problem domain language. For example, the situation model describes the actual code "pcboards = pcboards - sold;" as "reducing the inventory by the number of pc boards sold. This is done to keep an accurate count of inventory". In the same way that the program model consists of a hierarchy of chunked components, the situation model represents chunked plan knowledge. Lower-order plan knowledge can be chunked into higher-order plan knowledge. "The memory manager, the process manager, the secondary storage manager, the I/O system, the file manager, the protection system, networking, and the shell together define the operating system". This is the highest-order plan and it is comprised of lower-order plans containing knowledge about each component.

The mechanisms used for situation model building are the same as those used for program model building: cross-referencing and chunking. The only difference is that the knowledge involved is domain plan knowledge, as opposed to program model text-structure and plan knowledge.



Again, beacons can play an important part in determining which plans are used. The matching process takes information from the program model and builds hypothesized higher-order plans. These new plans are stored in long-term memory and are also chunked into higher-order plans. The situation model as a mental representation contains a functional and data-flow abstraction of the program.

Pennington's model is quite flexible. Information flows between the program and situation models illustrate the program model is modifiable after situation model construction begins. A cross reference map allows a direct mapping from procedural, statement level representations to a functional, abstract view of the program. Higher-order plans can cause a switch to program model building, either directly modifying the text-base or as input to the program model comprehension process.

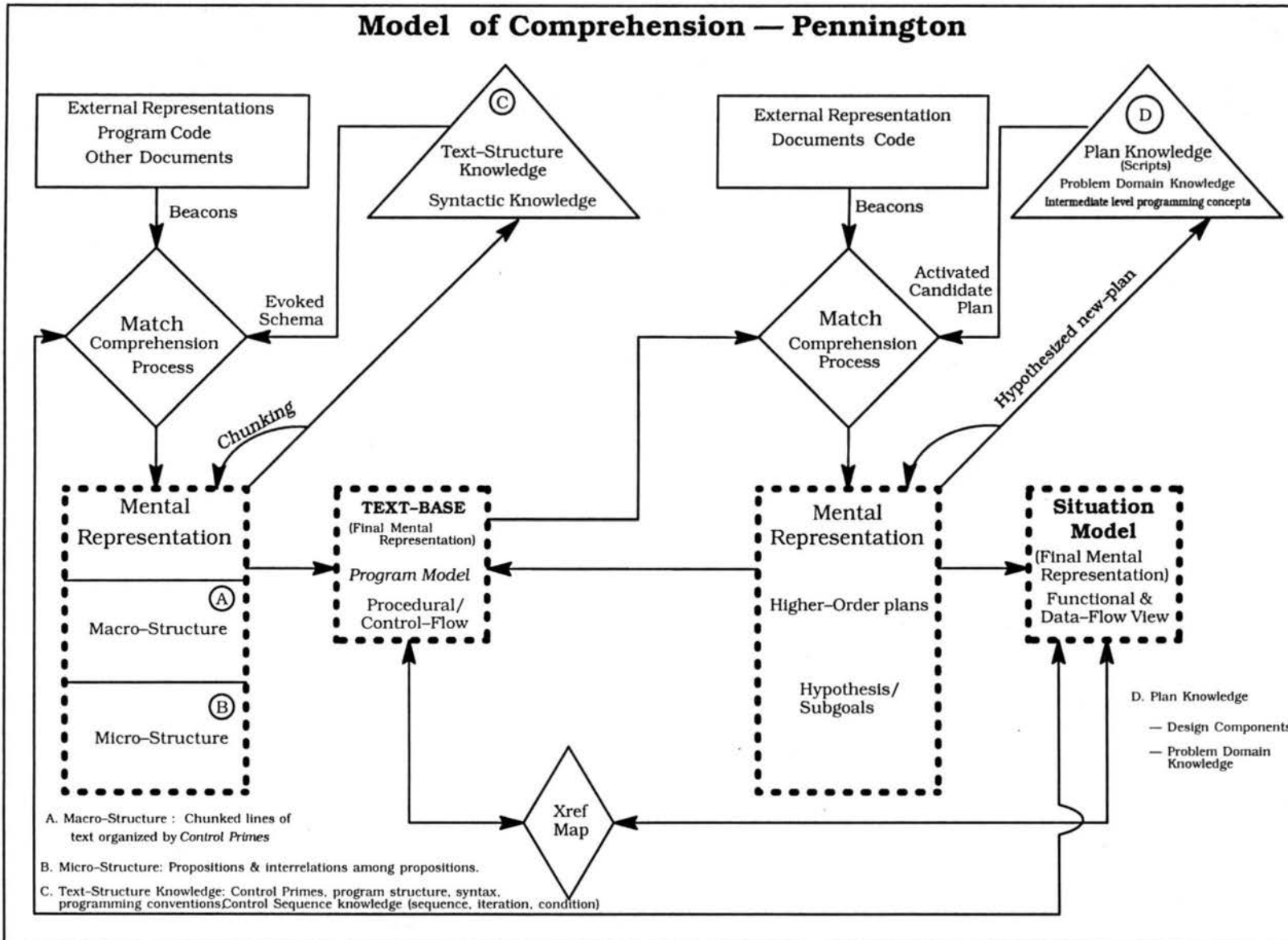


Figure 3.5: Pennington - Comprehension Model

### 3.3.6 Evaluation of Models

The five models have several commonalities. At the highest level of generality, all five accommodate 1). a mental representation of the code, 2). a body of knowledge (knowledge base) stored in long-term memory, and 3). a process for incorporating the knowledge in long-term memory with external information (such as code) into a mental representation. Each differs in the amount of detail for each of these three main components.

Letovsky's model is the most general code comprehension model. This model focuses mainly on the form of the mental representation. There are no details on how the assimilation process works or how knowledge is incorporated into the mental representation beyond the statement that it occurs. However, knowledge types are more detailed and coincide with Soloway & Ehrlich's model. Shneiderman's model is more detailed because it includes a hierarchical organization of knowledge and a separation between semantic and syntactic knowledge. Similar to Letovsky, the focus is on the form of the mental representation, but few details are provided on how construction is done.

Brooks' model is different from the other models in that all changes to the current mental representation occur as the result of an hypothesis. The mental model (for program understanding) is constructed in one direction only, from the problem domain to the program domain. The knowledge structures used are not well defined. Although hypotheses are important drivers of cognition, there are other ways of updating the current mental representations, for example strategy-driven (e.g. cross-referencing or systematic/opportunistic). Also, if understanding occurred in the direction indicated, it would not be possible to switch from one level of abstraction to another going in an opposite direction. For example, suppose an engineer was currently trying to understand a piece of code someone else wrote. This implies she will start building her mental representation from the right-hand side of Figure 3.3. If at some point she makes a connection to the domain, there is no way to jump back to the right-hand side of the figure. The Brooks' model is similar to the Soloway and Ehrlich model in that the mental representation is constructed top-down through finer levels of details.

Pennington's model is more detailed and includes specific descriptions of the processes and knowledge necessary to build two thirds of the entire mental model. It accounts for the types and composition of knowledge needed to construct most of the mental repre-

sentation as well as the form of the mental representation. It also contains mechanisms for abstraction. The major drawback of this model is the lack of higher level knowledge structures such as design or application domain knowledge.

Soloway and Ehrlich's model (also known as the domain model) emphasizes the highest level abstractions in the mental model. One aspect that sets this model apart from the others is the top-down development of the mental model with the assumption that the knowledge it uses has been previously acquired. By itself, this model does not take into account situations when code is novel and the programmer has no experience to use as a foundation for program comprehension.

Each of these models represent important aspects of code comprehension and many overlap in characteristics. For example, Brooks, Letovsky, and Shneiderman all focus on hierarchical layers in the mental representations. Brooks, Letovsky, and Soloway & Ehrlich use a form of top-down program comprehension while Pennington & Letovsky use a bottom-up approach to code understanding. Shneiderman's model uses both top-down and bottom-up understanding depending on whether the programmer understands the code or develops it. All five models use a matching process between what is already known (knowledge structures) and the artifact under study. No one model accounts for all the behavior we see when programmers understand unfamiliar code. However, we can take the best of these models and combine them into an *Integrated comprehension model* that not only represents relevant portions of the individual models but also behaviors not found in them, e.g. when a programmer switches between top-down and bottom-up code comprehension.

### **3.4 Integrated Model**

von Mayrhauser's and Vans' integrated code comprehension model [62, 63] consists of four major components, (1.) Top-Down model, (2.) Situation model, (3.) Program model, and (4.) Knowledge base. The first three reflect comprehension processes. The fourth is necessary for successfully building the other three. Each component represents both the internal representation of the program being understood (or short-term memory) as well as a strategy used to build this internal representation. The knowledge base either

Table 3.1: Code Comprehension Model – Levels of Detail

<i>Maintenance Task</i>	<i>Models</i>	<i>Level 1</i>	<i>Level 2</i>	<i>Level 3</i>
Adaptive Perfective Corrective Reuse Code Leverage	Top-Down Program Model Situation Model	Processes Hypotheses Strategies	Plans Rules of Discourse Chunks Episodes	Beacons Text Structure Action Types

furnishes the process with information related to the comprehension task or stores any new and inferred knowledge.

The integrated model combines the top-down understanding of [49] with the bottom-up understanding of [36], recognizing that for large systems a combination of approaches to understanding becomes necessary. Our observations showed that programmers switch between all three comprehension models [62, 63].

Any of the three sub-models may become active at any time during the comprehension process. For example, during program model construction a programmer may recognize a beacon indicating a common task such as sorting. This leads to the hypothesis that the code sorts something, causing a jump to the top down model. The programmer then generates sub-goals (e.g. I need to find out whether the sort is in ascending or descending order) and searches the code for clues to support these sub-goals. If, during the search, he finds a section of unrecognized code, he may jump back to program model building. Structures built by any of the three model components are accessible by any other, however, each model component has its own preferred types of knowledge.

### 3.5 Levels of the Integrated Model

The Integrated Code Comprehension Model has many levels that are not apparent in Figure 3.6 because the figure only represents connections between the component models and the knowledge they use. There are several levels of detail supporting each component in the integrated model as shown in Table 3.1.

Table 3.1 contains three levels of detail in order of decreasing abstraction. Each level contains elements of code comprehension which may be composed of sequences or

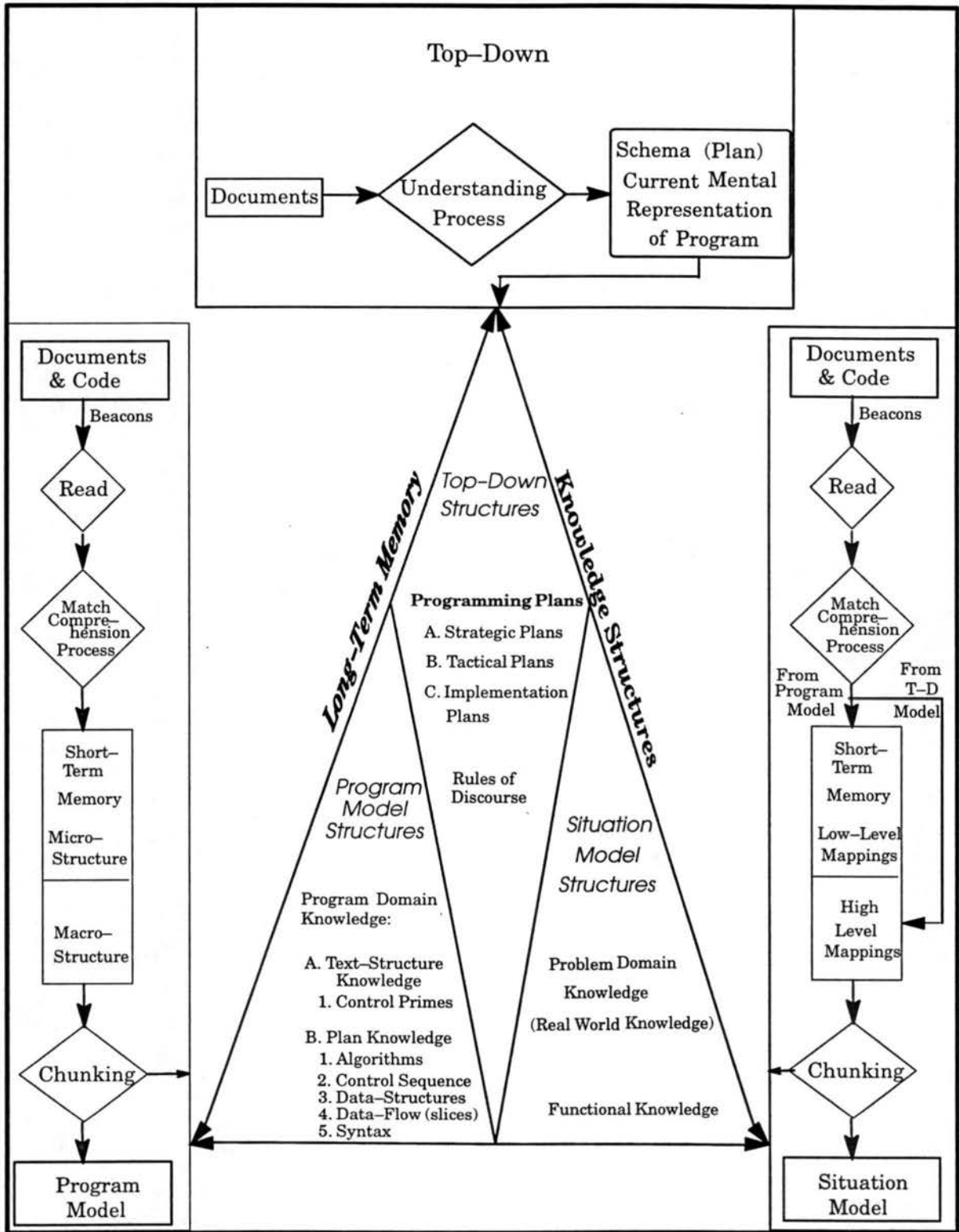


Figure 3.6: Integrated Code Comprehension Model

frequencies of elements in the next lower level. Components within a level may also be related.

At level 3 we find the lowest level of detail which includes *action types*, beacons, and text structure. Action types classify programmer activities, both implicit and explicit during a specific maintenance task. Examples of action types include “asking a question” and “generating an hypothesis”. Action types are important because they define *episodes* which are found at the next higher level (level 2) in the table. Episodes are composed of sequences of action types. At level 2, chunks are composed of sequences of level 3 text structure. Beacons evoke higher level plans and rules of discourse.

At level 2, sequences of episodes aggregate to form higher-level *Processes* at level 1. Plans, rules of discourse, and chunks govern strategies at level 1. Plans are related to rules of discourse since a plan may be evoked as a consequence of some rule. Plans are also related chunks because plans can include chunks in the definition of plans. Rules of discourse and previously evoked plans may trigger level 1 hypotheses.

At the highest level are the integrated model components. Processes, hypotheses, and strategies are the highest level dynamic elements found during top-down, situation, and program model construction.

### 3.6 Research Grid

Several levels of detail comprising a code comprehension model have been identified above. The experiments shown in Table 2.2 can be reorganized according to these levels. Table 3.2 shows the elements that have been studied, those that do not have much empirical research, or those that have been ignored.

Table 3.2 classifies experiments related to code comprehension according to these levels of detail. The columns indicate several characteristics of the experiments. For code size, *small* applies to programs of less than 900 lines of code (LOC). Medium size code refers to code between 900 and 40,000 LOC. Large scale code consists of more than 40,000 LOC. The language columns correspond to legacy languages such as Cobol, Fortran, Basic, and Pascal. The *UNIX/C/C++* language column applies to the C/C++ languages and environment programs and libraries like *Make*, *lint*, *Motif*, *lex/yacc*, *awk*, *perl*, etc. Subjects are categorized as novice, graduate students, or professional programmers. The

Table 3.2: Research Grid

Level	Size: Small	Size: Medium	Size: Large	Language: Legacy	Language: UNIX/C /C++	Subject: Novice	Subject: Grad Student	Subject: Profes- sional
Models								[3]
Top-Down				[36]				[36]
Situation	[36]			[36]				[36]
Program	[53],[29]			[53],[8], [29]		[53]		[53],[8], [29]
Other								
Level 1								
Processes	[29],[53]			[29],[53]		[53]		[29],[53]
Hypotheses	[18],[27], [30],[37], [47],[50]			[18],[27], [30],[37], [47],[50]		[47]	[18],[47]	[27],[30], [37],[47], [50]
Strategies								
Level 2								
Plans	[16],[17], [42],[46], [48]			[16],[17], [42],[46], [48]		[16],[42], [46],[48]	[16],[17], [42],[48]	[16],[19]
Rules of Discourse	[48]			[48]		[48]	[48]	
Chunks	[2],[7], [52]			[2],[7], [52]		[2]		[2],[7], [52]
Episodes	[53]			[53]		[53]		[53]
Level 3								
Beacons	[22],[67]	[23],[33], [34]		[22],[33], [34],[67]	[23],[33], [34]	[67]	[22],[33], [34],[67]	[23]
Text Structures	[36]			[36]				[36]
Actions	[29]			[29]				[29]

rows reflect the components and levels of understanding from Table 3.1. Each cell in the table represents the experiments that have investigated the row component with the column attribute.

Table 3.2 illustrates where research is clearly lacking. No experiments for any of the components have been investigated for large-scale code. Just two experiments have even used medium-sized code. The C language has been used in only two experiments while none used C/C++ programming environments or tools. The majority of the studies investigate strategies, beacons, or plans. There are no studies of processes, and only one study of rules of discourse, actions, and program comprehension episodes. Similarly, one study of each major comprehension model has been done. Thus, it is easy to see that a great deal of research is still needed. The current research focuses on professional programmers working on large-scale code using contemporary languages and environments. The two columns with the largest number of empty fields are (Size: Large) and (Language: UNIX/C/C++).



Experimental results of small scale code have not been shown to scale up for production code. Results of experiments on large-scale code comprehension will provide better support for tools, maintenance guidelines and processes, and documentation for more efficient and effective software maintenance.

This chapter explained *what* code comprehension is and that studies of production code have been ignored. To address this, we have designed a study of large-scale code comprehension that includes the use of UNIX, C, and C++. The next chapter, *Research Design*, contains a description of *how* we accomplished this research.

## Chapter 4

### RESEARCH DESIGN

This chapter discusses the design of the our study including a description of the subjects who participated, a detailed list of each of the study objectives, and an explanation of the data analysis used in determining the results.

#### 4.1 Experimental Design

The purpose of this study is to derive a theory for large-scale code comprehension using the Integrated Comprehension Model. Several observations of professional maintenance engineers doing various maintenance tasks have been collected. Each observation involved a *programming session* whereby the participants were asked to think aloud while working on understanding code. We audio- and video-taped each observation as a thinking aloud report. Sessions were typically two hours long. As this is not enough to understand a large scale software product, we found participants who had varying degrees of prior experience with the code. This allows the widest degree of coverage over the code comprehension process.

Eleven subjects have been video- and/or audio-taped. The tapes were then transcribed for analysis. The following list gives an identification code for each subject, a brief description of the task, and the subject's expertise. These codes are used in the rest of this document to identify individual subjects.

- AD1 : Port Program Across Platforms – Domain/Language Expert
- AD2 : Add Functionality, Prototype Assessment – Domain Expert, Language Novice
- C1 : Fix Reported Bug – Domain Novice, Language Expert
- C2 : Understand Bug – Domain Expert, Language Novice

- C3 : Fix Reported Bug – Domain Expert, Language Novice
- C4 : Track Down Bug – Domain Expert, Language Novice
- EN1 : Add Functionality – Domain/Language Expert
- EN2 : Track Down Bugs and Finish Adding Functionality – Domain/Language Expert
- G1: Program General Understanding – Domain/Language Expert
- G2: General Understanding, One Module – Domain/Language Expert
- L1: Understand Program for Leverage – Domain Novice, Language Expert

Table 4.1 defines three major variables for our study. The columns represent *expertise*, the rows represent the amount of accumulated knowledge subjects had acquired *prior* to the start of each observation. Although there are six rows containing descriptions of varying degrees of accumulated knowledge, these categories have been collapsed into three general groups: *Little*, *Some*, and *Significant*, and are indicated by italics in the table. The *type of maintenance task* is listed as an entry in the matrix. Each square represents specific observations that are characterized by the row, column, and maintenance task.

As the matrix shows (Table 4.1) these eleven subjects represent good (albeit not perfect) coverage in terms of expertise and varying degrees of knowledge about the software.

Similar to [24], we face three issues in determining the validity of generalizing results from our data to other maintenance situations. These issues deal with maintenance task, sampling of participants, and external validity of the experimental situation.

1. Task. The code the participants tried to understand and the specific assignment performed were representative of the maintenance tasks encountered in industry. These tasks include debugging, enhancements, code adaptations, code leveraging, and general understanding. The average size of the maintainable code is between 50,000 and 80,000 LOC with a few examples much larger than 80,000 LOC. While the participants were not all doing identical tasks, they were all trying to understand production code to maintain it.

Table 4.1: Programming Sessions – All Maintenance Tasks

<i>Expertise</i> ⇒ Accumulated Knowledge↓	<i>Language Expert</i>	<i>Domain Expert</i>	<i>Language &amp; Domain Expert</i>
Never Seen Before ( <i>Little</i> )		<b>C2:</b> Understand Bug	
File Structure Call Graph ( <i>Little</i> )		<b>C3:</b> Fix Reported Bug	<b>G1:</b> Program: General Understand <b>EN2:</b> Add Function
Requirement & Design Documents ( <i>Some</i> )	<b>C1:</b> Fix Reported Bug		<b>G2:</b> Under- stand one Module
Worked some with code, style familiar ( <i>Some</i> )	<b>L1:</b> Leverage Small Program	<b>C4:</b> Track Down Bug	
Prior code enhancement, debugging, adaptations ( <i>Significant</i> )		<b>AD2:</b> Add Function, Prototype Assess	<b>AD1:</b> Port Program across Platforms
Worked with code several years ( <i>Significant</i> )			<b>EN1:</b> Add Function- ality

As we begin to understand whether and how cognition differs, we can move to more specialized tasks to explore each situation further.

2. Sampling of participants. How representative are our participants of the larger population of software engineers who work with existing code? There is no reliable answer given the current maturity of the field. We attempted to sample a broad range of maintenance tasks, prior work with the code, and programmer experience in domain and language. We make no claim that these protocols represent the full range of cognition behavior of the population of software maintenance engineers. It is more likely that the description of this population will need to be assembled from many (future) studies similar to ours.

3. External validity. This concerns the degree to which the conditions used for collecting observational data are applicable to actual maintenance tasks. Code cognition for maintenance purposes takes more than two hours. We compensated for lack of task completeness by including different amounts of prior preparation in our study. This allows us to observe similar tasks at various stages of completion. All tasks represented actual work assignments. Together, actual tasks observed at different phases strengthen the generalizability of our findings.

## 4.2 Program Comprehension Study Objectives

### 1. *Action Types - Level 3*

What are the typical programmer actions executed during maintenance? Do common actions differ depending on the particular maintenance task? What are the most necessary actions for building the top-down, situation, or program mental model?

### 2. *Episodes - Level 2*

Are there repeated action sequences (episodes). How often do they occur? How similar are they? Which types of episodes occur most frequently? What information does the engineer need to complete an episode? Do episodes represent understanding at only one level of abstraction (i.e. top-down, situation, program model levels) or do they span all levels?

### 3. *Processes - Level 1*

*Aggregate Processes* - If we successfully find episodes, how will they be used in higher level understanding? Are there repeated episode sequences (aggregate processes)? How similar are they? What *triggers* the end of one episode and the beginning of another in a sequence? Are some of these triggers more common than others?

*Session-Level Processes* - Once we find the aggregate processes, how will they help in defining the overall maintenance task process (session-level)? Are there repeated sequences of aggregate processes? What distinguishes a switch from one aggregate process and another in the sequence? If we find one process for each type of maintenance task, what are their similarities?

#### 4. *Hypotheses – Level 1*

How are hypotheses used in the understanding process? Are there common types of hypotheses? How do they facilitate model construction?

#### 5. *The role of Model Components in the Integrated Model – Level 1*

Do expert maintenance engineers frequently switch between all model components (i.e. understanding is built at all levels of abstraction simultaneously)? Is there a difference in working at levels of abstraction based on the size of the code under consideration? Is there a difference between abstraction level focus based on maintenance task?

#### 6. *Information Needs and Core Competencies.* Once we find answers to the above questions concerning actions, processes, and hypotheses, can we organize the information engineers need during understanding to support cognition? Can we define tool capabilities that help support the way programmers actually understand code? Can we define core competencies that reflect the minimum knowledge necessary to maintain large scale code?

### 4.3 Protocol Analysis

Protocol analysis is a technique used for analyzing observation data. Think-aloud reports of subjects working on tasks are transcribed and classified using categories decided on prior to the actual analysis [20]. For example, we expect to find maintenance engineers generating hypotheses and reading code during maintenance. Thus, protocol analysis includes searching for these two activities. Specifically, each statement in the transcript is *encoded* as one of the a priori categories. Thinking aloud must occur concurrently with the task for the data to be accurate. If the data collected is introspective (i.e. after the task is complete) it is probable it will contain faulty details. Introspection may be useful for the discovery of psychological processes but it is worthless for verification [21].

For example, we ran an initial pilot study using an *interview* method in which we used a questionnaire to interview an expert maintenance engineer. The questions were categorized by integrated model components. For example, questions about syntax were

asked to gain insight into program model building. For example, *“How and at what point do you use sequence or iteration constructs to understand code?”* Questions about the situation model included questions like *“At what time during the code comprehension task do you draw on your knowledge of the objects and their interactions in the real world?”* We made an attempt to design the questionnaire in such a way as to draw answers from the interviewee without suggesting the answer. This attempt proved unsatisfactory and many times the engineer would give academic text-book answers. His responses were an indication that he was answering not from experience but from what he had been taught as the correct method.

The questionnaire used an introspective technique that is limited in the amount of detail and experimental control. However, this technique is a reasonable method for determining whether the integrated model is complete enough to proceed with more detailed think-aloud sessions and protocol analysis.

Protocol analysis is a method used to discover cognitive processes while the subject is performing some task. There are three general ways in which verbalizing information is related to task-directed processes. In the first, the task to be performed is not related to the information the subject is asked to verbalize. Second, verbalization is generated by the task. Finally, verbalization changes the task. This occurs when the verbalization requirements call for information that is not normally attended to during the task. The task can be affected when a subject is asked to verbalize motives and reasoning in situations where they do not typically pay conscious attention to this type of information. Figure 4.1, taken from [21], illustrates the second verbalization task.

The second type of verbalization task elicits the information needed for this research. Since the task determines the type of information concentrated on and what is verbalized, this is the process we wanted the 11 subjects to use during the think aloud sessions. Subjects were instructed to verbalize what came into their heads as they worked on their maintenance task. The interviewer intervened only to remind subjects to keep talking. According to Ericsson and Simon [21], these think aloud reports are considered real data and protocol analysis is an appropriate analysis technique for them.

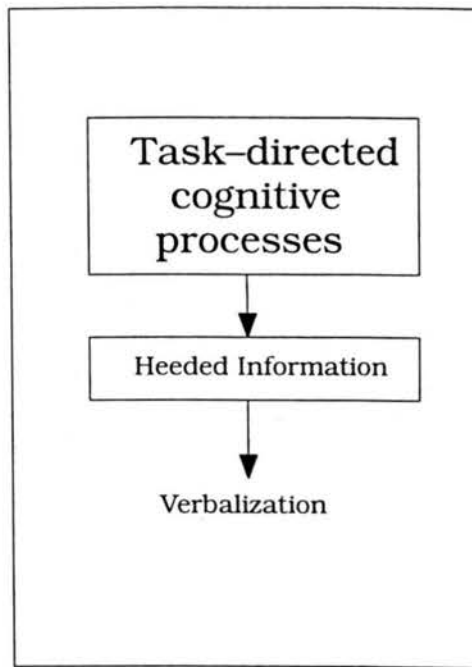


Figure 4.1: Verbalization Task

Two major types of analysis are possible for the coded think aloud reports. A *frequency* analysis determines how often each type of code (e.g. X) appears in the protocol and a *sequence* analysis determines if there are common sequences of codes (e.g. X followed by Y). Frequencies indicate the existence of X (a static property) and sequences demonstrate the dynamic aspects (i.e. how is X used).

The two major problems with protocol analysis are that the encoding may not be objective and that thinking aloud might alter the activity itself. One way of making sure the encoding is objective is to apply a *reliability analysis* wherein (a portion of) the protocols are analyzed by at least two unbiased judges and the results are then compared. If the two judges agree a high percentage of the time, then the encoding is considered reliable. The typical "percentage agreed" is around 90%. Several studies [20] have shown that thinking aloud while performing a task slows down the task but does not significantly alter it unless the subject is required to verbalize information not normally needed.

Protocol analysis proceeded in five steps (see Table 4.2). The following subsections describe in more detail the criteria to categorize statements, identify information needs, and analyze protocols for discovery of processes, strategies, and hypotheses.



Table 4.2: Protocol Analysis Steps

<i>Enumeration</i>	<i>Segmentation</i>	<i>Process Discovery</i>	<i>Strategy Identification</i>	<i>Action Type Analysis</i>
Utterance ↓ Action Types	1. Abstraction level 2. Action types 3. Information Needs & Tool Capabilities	1. Episode level processes 2. Aggregate level processes 3. Session level processes	Processes ↓ Strategies	1. Hypothesis Generation 2. Switching Behavior

#### 4.4 Enumeration and Segmentation of Action Types – Level 3

The first analysis on the protocols involve *enumeration of action types* as they relate to the top-down, situation, and program models. Transcripts of the protocols were analyzed using a list of expected actions developed by Vessey [53]. New action types were discovered during this process and subsequent protocols were analyzed for the new actions as well. Appendix A contains a list of action types. A description of the types of statements that comprise the actions along with key words and phrases are included. The next step in the analysis segments the protocols. Segmentation classifies action types into those involving domain (top-down), situation, or program model and can be thought of in terms of different levels of abstraction in the mental model.

Protocol analysis is an iterative process. A first pass analysis results in a high-level classification of programmer actions as either top-down, situation, or program model components of the Integrated Model. This is necessary because similar actions appear in different component processes. For example, hypotheses may be generated while constructing any of the three models. Once actions are associated with a particular model component, the next pass identifies action types of a specific maintenance task. Once the action types are identified, the transcripts are re-analyzed and encoded using these types as tags on the programmer utterances. <sup>1</sup> Table 4.3 contains example protocols to show action type classification. Column two provides the tag used in action type classification.

---

<sup>1</sup>Utterances are verbalizations of programmers during programming sessions and captured in the transcripts.

Table 4.3: Example Protocol Analysis – Action Types

<i>Analysis Type</i>	<i>Tag</i>	<i>Action Type</i>	<i>Example Protocol</i>
Action-Type Classification	SYS8	<b>Generate Hypothesis (Program Model)</b>	"...and my assumption is that nil with a little <i>n</i> and nil with a big <i>N</i> are equivalent at the moment."
	SYS7	<b>Chunk &amp; Store knowledge (Program Model)</b>	"So clearly what this does is just flip a logical flag"

Frequency analyses can be done to answer the first set of questions (number 1) in section 4.2. Counts of actions from individual protocols can be aggregated by maintenance task, then for all maintenance tasks together, and finally by model component (i.e Top-down, situation, and program model).

#### 4.5 Process Discovery – Level 2 & Level 1

We can discover *dynamic* code understanding *processes* by classifying and analyzing *episodes* [53]. Episodes are single instances of a sequence of action types. An episode starts with a *goal* and embodies the actions to accomplish that goal. For example, determining the function of a specific procedure or routine may entail a sequence of steps that include reading comments, following control flow, and generating questions when a concept is not understood. Similar to Vessey [53], *processes* are defined at three different levels; *episodic*, *aggregate*, and *session* levels. Episodes containing common action types with similar goals are defined as a single episodic process. Likewise, common sequences of episodic processes are defined as a single aggregate level process. Finally, the session level process is established by a sequence of similar aggregate level processes.

A process is a sequence of action types, episodes, or aggregates whose purpose is to satisfy a specific goal. We can think of them as three levels of strategies to achieve goals. Specifically, each episode is determined by discovering the goal (or sub-goal) and cataloging all subsequent action types until reaching closure on the goal (due to being satisfied or abandoned). If specific information is needed during the episode, this is noted as an *information need*. Once episodes are identified, we analyze each, abstract out commonalities, and designate the resulting sequence an episodic process. An aggregate level process emerges from similar episodic processes. Similarly, sequences of aggregate processes are an-

alyzed for commonalities and abstracted into higher-level processes. Once again, common sequences of aggregate level processes produce a single session level process representing a two hour programming session.

Frequency counts and analyzing for sequences helps to answer the episode questions in study objective number 2. To determine if there are repeated action sequences and how often they occur, we look for common sequences of action types and then count the number of occurrences of each episode. Information needs associated with a particular type of episode can also be counted. By examining sequences of action types and their related model components, we will be able to determine if individual episodes occur at only one level of abstraction or span all three levels.

The third study objective question concerns aggregate level processes. Again, we can use frequency counts and sequence analysis. Sequence analysis on episodes defines aggregate processes. A count of these sequences determines the most common aggregate processes. If one aggregate process contains many of the same episodes as another aggregate process, we consider them to be similar. The action types and information needs that occur at the beginning or the end of an episode sequence are process boundaries called triggers. We can count these triggers to discover the common types. Once aggregate processes are defined, we can analyze them for common sequences to distinguish session-level processes. Counting triggers between aggregate processes will distinguish common actions and information needed to switch from one process to another. Finally, comparing session-level processes will indicate the degree of similarity between processes for different maintenance tasks. Processes that contain many of the same aggregate level processes are considered comparable.

#### **4.6 Action Type Analysis: Hypotheses & Switching Between Models**

Two types of actions are important enough to motivate a more detailed analysis, these are hypotheses and switching between integrated model components. Making hypotheses serves a special role because hypotheses help to guide investigation during code cognition. Switches are significant because they indicate a change of focus from one of the three integrated model components to another. Additionally, hypotheses and switches are related in that hypotheses may be one of the causes for switching between model components.

Table 4.4: Example Protocol Analysis – Hypotheses

<i>Analysis Type</i>	<i>Tag</i>	<i>Hypotheses Type</i>	<i>Example Protocol</i>
Hypotheses-Classification	SYSH1	<b>Variable Function (Program Model)</b>	"...looks to me like TMODE (...) is used essentially to determine whether or not (..) we're echoing back to the screen"
	OPH3	<b>Rules of Discourse Expectations (Program Model)</b>	"I would be surprised if that's not called "somewhere else"

#### 4.6.1 Hypotheses – Level 1

Hypotheses are associated with model components. They are initially identified during action type analysis. During action type analysis, we tag generation, confirmation, or failure of each hypothesis. Similar to action-type analysis, each hypothesis is then classified using a set of apriori categories in the protocol. For example, the utterance “warnings are related, we believe, to the resources either not being installed properly or not being compatible with this system,” is classified as a hypothesis about the cause of the buggy behavior. Table 4.4 contains example protocols to illustrate hypothesis type classification.

A separate analysis follows each hypothesis through the protocol until it is resolved through confirmation, failure, or abandonment. A hypothesis is verbally confirmed or rejected (i.e. it fails). Hypothesis abandonment can either be explicit or implicit. Explicit abandonment occurs when the programmer decides it is not relevant or will be too much trouble to verify. Implicit abandonment occurs when the hypothesis is stated but the programmer never returns to it. We can only conjecture that the hypothesis was either forgotten or dismissed without verbal confirmation.

Each major hypothesis is generated as a result of some goal. The goal itself is resolved when the underlying hypotheses are confirmed, abandoned, or fail. An analysis similar to that for hypothesis resolution was performed. We also analyzed the dynamic behavior of goal statement to goal completion over time. This results in a hierarchical representation of each goal, with its supporting hypotheses and actions. It can also provide a transition diagram illustrating temporal relationships of goals.

To answer the questions in study objective 4, analysis can proceed by examining each hypothesis, determining its type within the integrated model, obtaining a frequency for each type, and comparing them to the hypotheses found during other maintenance tasks.

#### 4.6.2 Switching between Model Components – Model Level

A switch occurs when any action in the top-down, situation, or program model is followed by another action in a different model. For example, an hypothesis generated after reading a block of code (program model) may be followed by concluding the functionality of that block (situation model). Switching behavior helps to verify that the integrated model is valid when there is a high frequency of switches between model components. The first question in study objective 5 can be answered by counting the number of switches between each integrated model component. To determine if there is a difference in working at levels of abstraction based on code size we compare the total number of integrated model switches between subjects that worked on code of various sizes. We can compare the number of switches for each maintenance task to learn if task changes the frequency of switches between model components.

An hypothesis is an action that can generate a switch from one model component to another. We looked specifically at switching caused by hypotheses. An hypothesis generated switch is identified by examining the action immediately following each hypothesis. If the action occurs at a different level of abstraction than the generating hypothesis, then a switch is recorded. Examining switching behavior based only on hypotheses will give an indication of the relative importance of hypotheses in building mental representations at all levels of abstraction.

#### 4.7 Strategy Identification – Level 1

Strategies guide the sequence of actions during goal satisfaction. Our analysis involves looking for two different types of strategies. The first type is the overall strategy used for understanding tasks. This is usually based on the programmers *preference* or *style* of understanding. An example is a systematic versus an opportunistic style. A second type of strategy is associated with mental model building. Pennington [37] found that expert programmers used a *cross referencing* strategy between the program model and the situation model while understanding a small program. This cross referencing strategy can be thought of as chunking program model information and using a situation model tag to abstract the newly acquired information.

It is possible that more than one strategy may be in effect concurrently. For example, an overall systematic strategy can occur at the same time as cross-referencing since one is independent of the other. One is a method for the task of reading code while the other is a method for building an understanding of the code.

Switches may also be related to strategies, for example the cross referencing strategy involves switches between the program and situation models. Switches need to be studied to determine if other cross-referencing strategies (e.g. between program and top-down or between situation and top-down model) are common during program comprehension.

Examination of the overall session-level process will indicate if the programmer used a systematic or opportunistic approach to understanding. An opportunistic strategy will be apparent if only very relevant sections of code were investigated. If code was studied line by line, then the systematic strategy was applied. It is also possible that a combination of opportunistic and systematic strategies will be used. To determine which type of cross-referencing strategies are used, frequency counts for switches between integrated model components provide the answer. If the majority of switches occur between program and situation model, then the strategy is a program-situation model cross-referencing strategy.

#### **4.8 Information Needs**

Information Needs are determined from protocols directly from the transcribed tapes (see Table 4.5) or through inference. These needs may be explicit or implicit. An implicit information need is not directly stated but the programmer could obviously profit from it if he knew it existed. For example, in Table 4.5 the protocol segment associated with the *Code Block Boundaries* information need indirectly demonstrates that the related domain information would help this programmer understand the code better. Indeed, he spent a great deal of time examining several documents for this very information. Table 4.5 contains example protocols to show information needs identification.

Common information needs can be tallied and translated into *tool capabilities* which describe the functions that should be available for tools to address the information need. Information needs can be based on maintenance task by determining how often the information is needed during the particular task type. Finally, the most common types

Table 4.5: Example Protocol Analysis – Information Needs

<i>Analysis Type</i>	<i>Information Need Classified As</i>	<i>Example Protocol</i>
Identifying Information Needs	<b>Code Block Boundaries</b>	“Okay well, assuming that the ...um indentation is accurate...I would guess that this is really for, um, there must be a FOR statement that this is the end of but I don't know where that FOR statement might come from. I don't see...”
	<b>Data structure tied to concepts in the domain</b>	“And this looks like some X.25 structure”

of information required can be translated into guidelines that prescribe the minimum knowledge necessary to maintain large-scale code.

#### 4.9 Summary

The objective of this study is to construct a theory of large-scale code comprehension using the Integrated Model. To do this, action types, processes, hypotheses, switching behavior, strategies, and information needs were investigated for maintenance engineers working on production code.

Think aloud reports of 11 engineers working on various maintenance tasks have been collected for the types of analysis described in this chapter. This analysis reveals common types of behaviors and information necessary for successful completion of specific tasks and maintenance in general. These results can be used to abstract and describe the minimum knowledge necessary to perform maintenance tasks. This knowledge, which we call *core competencies*, can be used for maintenance guidelines and processes, documentation standards, and tools.

## Chapter 5

### RESULTS

This chapter summarizes and interprets individual results from analysis of data from eleven programming sessions. The purpose of this chapter is to present hypotheses derived from aggregate results.

Section 5.1 contains an analysis of action types by task, expertise, and accumulated knowledge. Comprehension processes are presented in section 5.2. Section 5.3 specifies hypotheses results, including common hypothesis types, switching behavior generated by hypotheses, and the HK-Ratio, which is a indicator of expertise based on hypothesis generation and use of previously acquired knowledge. Section 5.4 reports on overall switching behavior and the influence of code size on integrated model development. Information Needs, Tool Capabilities, and Core Competencies are presented next. We conclude with the most important hypotheses identified during this research.

Appendix B contains additional detail on results. Sections in the appendix parallel the outline of this chapter. Chapter Five aggregates data from individual subjects while more detailed analysis on subject groups is reported in Appendix B . Tables and graphs summarizing the results include task counts and sequences, model references and switches between models, hypotheses types and sequences, information needs, and tool capabilities.

#### 5.1 Actions – Level 3

This section presents results of the action-type analysis for all subjects. Action-types are found at the 3rd level of detail below model components. (See Table 3.1.) Results include the total number of references and normalized percentage for actions identified within each model. Normalized percentages for each model by task-type, expertise, and accumulated knowledge can be found in Appendix B. In this section we present the conjectures derived from data in Appendix B. The purpose of this analysis is to determine



the most common types of actions engineers perform during specific maintenance tasks. We are also interested in the influence of expertise and amount of accumulated knowledge on the types of actions commonly seen.

### 5.1.1 Action Frequencies Across All Tasks

Tables 5.1, 5.2, and 5.3 show counts for each of the top-down, situation, and program model action types for all subjects combined. The tables are organized according to total number of references, with the most frequently referenced action appearing at the top. The *Percent of TTL* indicates the percent of total action counts for the top-down model.

Table 5.1: Action Counts Across Tasks – Top-Down Model

<i>Code</i>	<i>Action Type</i>	<i>Total References</i>	<i>Percent of TTL</i>
OPKNOW	Use of Top-down knowledge	311	22%
OP3	Generate or revise hypothesis about functionality	242	17%
OP20	Generate task	213	15%
OP6	Determine understanding strategy	97	7%
OP13	Study/Initiate program Execution	92	6%
OP2	Determine next program segment to examine	87	6%
OP17	Chunk & store knowledge	83	6%
OP1	Gain High-Level Overview of Program	69	5%
OP15	Generate questions	55	4%
OP4	Determine relevance of program segment	49	4%
OP11	High-level change plan/Alternatives	37	3%
OP12	Observe buggy behavior	18	1%
OP22	Examine results of execution	15	1%
OP16	Answer questions	14	1%
OP14	Compare program segments	11	1%
OP18	Change directions	10	1%
OP9	Mental Simulation	7	0%
OP7	Investigate Oversight	6	0%
OP5	Determine if this program segment needs detailed understanding	2	0%
OPABAND	Abandoned hypothesis	113	47%
OPCONF	Confirmed Hypothesis	97	40%
OP8	Failed Hypothesis	32	13%

The most commonly found action during top-down model construction is use of top-down knowledge. Top-down knowledge is previously acquired knowledge that is elicited from long-term memory. Frequent use of top-down knowledge is support for the conjecture that top-down model development is the main focus when the programmer is familiar with the application domain [9, 49]. It is much easier to decompose the program into functional units if the programmer is well acquainted with the domain. Hypothesis generation (OP3)

(the second most common action found in the protocols) drives the decomposition process. Generating tasks (OP20) is the third most frequent action-type found in the protocols. The percent of total top-down references for these three types are 22%, 17%, and 15%, respectively. After these, there is a significant gap: the next most frequent type of top-down action is OP6 (Determine understanding strategy), with 7% of all actions. This indicates that the first three are more important than the remaining action-types.

In an analysis of comprehension processes we found a common pattern in opportunistic understanding for goal resolution consisting of a sequence of goal-hypothesis-action [55] (also see section 5.2.2.2). This means that a goal is resolved by making an hypothesis and then taking some action that results in the confirmation, failure, or abandonment of the hypothesis. Many of the actions involve applying knowledge and generating a new task (OP20). Thus, one theory for why the first three top-down action types were *using top-down knowledge*, OP3, and OP20, is that they are important for goal resolution and that goal resolution is critical for top-down understanding.

The last three rows of the table show how the hypotheses reported as OP3 were resolved. The majority were either confirmed or abandoned. Very few actually failed. The majority of the top-down hypotheses were generated by domain experts. Vessey [53] found that experts tend to pursue hypotheses that are easy to confirm and abandon those that are time consuming. More will be said about hypotheses in section 5.3.

Program model building typically involves generating an hypothesis (SYS8), examining code (SYS3), chunking/storing knowledge (SYS7), and *use of previously acquired program knowledge*. These represent the top four activities found for program model construction. (See Table 5.2.) There were many sequences involving these actions and several are embodied in episode-level processes (see section 5.2.1.1). Use of program model knowledge in the current mental model representation is important because knowledge that is stored becomes knowledge used in subsequent understanding activities. The top four program model action-types represent the most frequent cluster. They total 55% of all actions types at the program model level. The frequencies of the remaining ones are at most half of these four.

Table 5.2: Action Counts Across Tasks – Program Model

<i>Code</i>	<i>Action Type</i>	<i>Total References</i>	<i>Percent of TTL</i>
SYS7	Chunk & store knowledge	263	17%
SYS3	Examine next module in sequence	243	15%
SYSKNOW	Use of Program model knowledge	185	12%
SYS8	Generate Hypothesis	175	11%
SYS11	Generate new task	102	6%
SYS1	Read introductory code comments any related documents	93	6%
SYS10	Determine understanding strategy	89	6%
SYS2	Determine next program segment to examine	69	4%
SYS15	Generate/consider alternative code changes	50	3%
SYS12	Generate Question	48	3%
SYS4	Examine next module in control-flow	46	3%
SYS17	Add/Alter Code	43	3%
SYS21	Mental Simulation	43	3%
SYS5	Examine Data structures and definitions	29	2%
SYS20	Determine error/omitted code to be added	27	2%
SYS18	Determine location to set breakpoint	17	1%
SYS24	Search for begin/end of block	16	1%
SYS13	Determine if looking at correct code	14	1%
SYS23	Search for var definitions/use	12	1%
SYS16	Answer question	9	0%
SYS9	Construct Call Tree	3	0%
SYS14	Change direction	3	0%
SYS22	Compare code between versions	2	0%
SYS6	Slice on Data	0	0%
SYSCONF	Confirmed Hypothesis	75	43%
SYSABAND	Abandoned Hypothesis	69	39%
SYS19	Failed Hypothesis	31	18%

The last three rows in the table (SYSCONF, SYSABAND, SYS19) report on the resolution of program model hypotheses. Similar to top-down hypotheses, the majority of the hypotheses were either abandoned or confirmed.

The most frequent action in the situation model is *use of situation model knowledge* (cf. Table 5.3). Previously obtained situation knowledge, assimilation of new knowledge (SIT4), and hypothesis generation (SIT7) together drive the acquisition of new knowledge and help to organize it as chunks of information. The top three action-types embody 72% of the total number of situation model actions. These three actions and three of the four top action-types found for the program model are similar. This is important because Pennington [36] claims that programmers build a situation model after program model construction. Regardless of timing, it appears that the same types of frequent actions found in the program model occur at the situation model level. Whether mental model construction is uni-directional (i.e. first the program model is constructed, then the

Table 5.3: Action Counts Across Tasks – Situation Model

<i>Code</i>	<i>Action Type</i>	<i>Total References</i>	<i>Percent of TTL</i>
SITKNOW	Use of Situation model knowledge	219	28%
SIT4	Chunk & Store	208	26%
SIT7	Generate Hypothesis	144	18%
SIT1	Gain Situation Model knowledge	78	10%
SIT2	Develop Questions	38	5%
SIT6	Determine next info to be gained	25	3%
SIT8	Determine understanding strategy	25	3%
SIT5	Determine relevance of situation knowledge	21	3%
SIT11	Mental Simulation	21	3%
SIT3	Determine answers to questions	12	1%
SIT9	Determine if error exists (missing functionality)	3	0%
SIT12	Compare functionality between two versions	2	0%
SITCONF	Confirmed Hypothesis	65	45%
SITABAND	Abandoned Hypothesis	64	44%
SIT10	Failed Hypothesis	15	11%

situation model, as Pennington's results show for very small programs) or bi-directional, as our model indicates, will be discussed later when we analyze Switching Between Models.

The last three rows in the table (SITCONF, SITABAND, SIT19) report on the resolution of the situation model hypotheses. Similar to top-down and program model hypotheses, the majority of the hypotheses were either abandoned or confirmed.

The main types of actions across maintenance tasks for all three models include use of previously acquired knowledge, generating hypotheses, and chunking/storing of newly acquired knowledge into the mental representation. For all models, the spread between the top three or four action-types and the remaining actions is significant, which adds support to the hypothesis that these top action-types are important regardless of task.

While specific action types clearly predominate overall, it is also important to consider patterns by task type, expertise, and accumulated knowledge to identify whether actions are fundamentally different for various types of expertise and accumulated knowledge. We discuss this in the next section.

### 5.1.2 Actions – Model Frequencies by Task, Expertise, & Accumulated Knowledge

This section contains results based on action frequencies for each model. The results are organized by the task subjects were concerned with, the level of expertise, and

the amount of knowledge subjects accumulated prior to the programming session. (See Table 4.1 to determine the categories for each subject.) Table 5.4 shows a breakdown by model of total action frequencies and a normalized percentage for each subject. Percentages are calculated for each subject by dividing that subject's total actions across models by the number of actions in each model. Note that these numbers do not reflect abandoned or confirmed hypotheses numbers reported in the previous section. This is because during analysis of hypothesis resolution, whether an hypothesis was abandoned or confirmed could encompass many actions or no actions (i.e. in the case of forgetting the hypothesis was ever made). Thus, both abandoned and confirmed hypotheses do not fall nicely into an action category. Appendix B contains more detailed information on results, including bar charts which are provided to graphically depict the data reported in the table. These charts are organized according to task, expertise, & accumulated knowledge categories and represent the percentage of actions for each subject (in the set) by model. For example, Figure B.1 shows the percentage of top-down, situation, and program model actions for the subjects who were working on adaptive maintenance tasks, in this case, subjects **AD1** and **AD2**.

#### **5.1.2.1 Action Percentages By Task**

Table 5.4 contains a column for combined Program and Situation model references, roughly corresponding to Pennington's [36] comprehension model. We wanted to identify patterns based on differences between Pennington's bottom up model and the top-down model. Analysis of this data revealed several classes of subjects distinguished by percent of total top-down actions and combined program and situation model actions. These classes are:

- 1. {**AD1,AD2**}

Both subjects in the first class had the highest percent of top-down model references (67%) with a significantly lower percent of program and situation model references (33%). Obviously what makes these subjects similar was that they were both working on adaptation tasks and more specifically porting tasks. They were both domain

Table 5.4: Action-Types by Model – Frequencies & Percentages

<i>Subject Code</i>	<i>Top-Down Model</i>	<i>Program &amp; Situation Model</i>	<i>Situation Model</i>	<i>Program Model</i>	<i>Total Actions</i>
<b>AD1</b>	226	111	43	68	337
<i>Adaptation</i>	67%	33%	13%	20%	
<b>AD2</b>	220	108	47	61	328
<i>Adaptation</i>	67%	33%	14%	19%	
Total	446	219	90	129	665
<i>Adaptation</i>	67%	33%	14%	19%	
<b>C1</b>	46	303	160	143	349
<i>Corrective</i>	13%	87%	46%	41%	
<b>C2</b>	119	240	91	149	359
<i>Corrective</i>	33%	67%	25%	42%	
<b>C3</b>	119	202	42	160	321
<i>Corrective</i>	37%	63%	12%	51%	
<b>C4</b>	171	186	101	85	357
<i>Corrective</i>	48%	52%	28%	24%	
Total	455	931	394	537	1386
<i>Corrective</i>	33%	67%	28%	39%	
<b>EN1</b>	32	251	58	193	283
<i>Enhancement</i>	12%	88%	20%	68%	
<b>EN2</b>	154	276	53	223	430
<i>Enhancement</i>	36%	64%	12%	52%	
Total	186	527	111	416	713
<i>Enhancement</i>	26%	74%	16%	58%	
<b>G1</b>	222	190	78	112	412
<i>Understand</i>	54%	46%	19%	27%	
<b>G2</b>	59	276	33	243	335
<i>Understand</i>	17%	83%	10%	73%	
<b>L1</b>	50	234	90	144	284
<i>Understand</i>	18%	82%	32%	50%	
Total	331	700	201	499	1031
<i>Understand</i>	32%	68%	20%	48%	

experts and therefore had the knowledge necessary to work at the top-down model level. They also had similar levels of prior experience with the code.

- 2. {C4,G1}

The second class of subjects have roughly equal numbers of top-down and combined program and situation model references. What makes these two subjects similar is that both were domain experts and they took a similar approach to understanding the code they were working on. **G1**'s task was to look at a system for which he was recently given responsibility, while **C4** was trying to track down a networking bug in a system in which he had some (but not significant) experience. Both looked at code with which they were unfamiliar (**C4** spent significant time looking at system

library code he had not seen before) and they both examined actual code on a “need to know” basis.

- 3. {C2,C3,EN2}

All three subjects in the third class were either working on a debugging task or trying to track down bugs in recently enhanced code. These engineers had significantly more program and situation model references (between 63% and 67%) than top-down references (between 33% and 37%). The main influence here is probably task as finding bugs requires one to spend significant time looking at actual code. This group had exactly the opposite number of references to top-down and program/situation model actions than the first class of subjects.

- 4. {C1,EN1,G2,L1} The last class of subjects had considerably more program and situation model references (between 82% and 88%) than top-down references (between 12% and 18%). Two of the subjects, C1 and L1 were language only experts. It is difficult to build a top-down model when there is a lack of domain knowledge. Subject G2 used a systematic understanding strategy where he read the code line by line. EN1 also spent a great deal of time in the program model because he was interested in very specific parts of the code he had recently enhanced. Since he was a domain and language expert and had significant experience with the code, we surmise that his task did not require building a top-down mental model. He had worked with this code for several years and we assume already had a working top-down representation of the code. The task he was working on did not require much adjustment to the top-down mental representation.

Appendix B contains charts and more detailed explanations of the results presented here. As the four graphs in the appendix show, we can not make definite conclusions about the influence of task on the frequency of the actions associated with Top-Down, Situation, or Program Model building. Only Adaptive maintenance had significant similarities in percentages within each model. While we can not say much about percentages, we have seen that task may affect the trend between models. Except for one subject (G1) General Understanding, Adaptive, and Enhancement tasks all seemed to have the same spread across models in terms of relative frequencies of references to program, situation, and

top-down models. Corrective maintenance showed the same trend for 3 of the 4 subjects in that classification.

The situation model seems the least important for mental model construction. This is most likely related to the types of systems our subjects worked with. The majority of the software they worked with was system-level, e.g. operating system, network protocols, etc. The situation model is best described as the connection to the "real world" and system-level software has very few concrete connections outside the computer box.

Other factors such as expertise, accumulated knowledge, and overall strategy may mask task as an influence. Obviously, to make definite conclusions regarding the effect of task on model actions requires a larger sample size and control for expertise, accumulated knowledge, and strategies.

To summarize, we propose the following hypotheses as a result of this analysis:

- *Adaptation* has notable domain model action needs.
- *Corrective* maintenance has more top-down and program model than situation model action needs.
- *Enhancement* requires working closely with the code (program model).
- *General Understanding* behavior is varied, determined by strategy and prior knowledge.

Several of our observations point to expertise as having an effect on programmer actions. Grouping the data by level of expertise rather than task might provide hypotheses on preferred action-types.

#### **5.1.2.2 Action Percentages By Expertise**

Three levels of expertise were identified among the subjects: Language Only {C1,L1}, Domain Only {C2,C3,C4,AD2,}, and Language & Domain {G1,G2,EN1,EN2,AD1}. Table 5.5 organizes the relative frequency data by these groups. Additional detail can be found in Appendix B, including charts that depict graphically the percentage numbers reported in the table.



Table 5.5: Expertise: Top-Down & Program/Situation Model Relative Frequencies

<i>Expertise</i>	<i>Subject Identifier</i>	<i>Top-Down Model %</i>	<i>Program &amp; Situation %</i>
<i>Language Only</i>	<b>C1</b>	13%	87%
	<b>L1</b>	18%	82%
<i>Domain Only</i>	<b>C2</b>	33%	67%
	<b>C3</b>	37%	63%
	<b>C4</b>	48%	52%
	<b>AD2</b>	67%	33%
<i>Language &amp; Domain</i>	<b>G1</b>	54%	46%
	<b>G2</b>	17%	83%
	<b>EN1</b>	12%	88%
	<b>EN2</b>	36%	64%
	<b>AD1</b>	67%	33%

The subjects with no domain expertise limited their mental representation construction to program and situation models. They spent less than 20% of their actions in the domain model. Task seems to have less of an impact since the subjects were working on very different things. As domain knowledge is acquired, task seems to be more important in terms of the relative amount of time spent in specific models. There is indication of a trend toward more top-down references when domain knowledge exists. Domain only experts show at least twice the top-down references as the language only experts. Domain and language experts show more mixed behavior. Three of the five Domain and Language experts (**EN2**, **G1**, and **AD1**) show behavior similar to Domain Only experts while two (**G2**, **EN1**) show behavior closer to the Language Only experts. The similarity of references for **AD1** and **AD2** and **C2**, **C3**, and **C4** suggests that task may have an influence. To validate these conclusions requires an experiment in which language and domain expertise is varied while task and prior exposure to the code is held constant. If similar patterns emerge, the effect is clear. If not, task may have a more profound influence than domain and language and an experiment should be designed where task is varied and the other two variables are held constant.

### 5.1.2.3 Action Percentages By Accumulated Knowledge

Table 5.6: Accumulated Knowledge: Top-Down & Program/Situation Model Relative Frequencies

<i>Accumulated Knowledge</i>	<i>Subject Identifier</i>	<i>Top-Down Model %</i>	<i>Program &amp; Situation %</i>
<i>Little</i>	<b>C2</b>	33%	67%
	<b>C3</b>	37%	63%
	<b>G1</b>	54%	46%
	<b>EN2</b>	36%	64%
<i>Some</i>	<b>C1</b>	13%	87%
	<b>L1</b>	18%	82%
	<b>G2</b>	17%	83%
	<b>C4</b>	48%	52%
<i>Significant</i>	<b>AD1</b>	67%	33%
	<b>AD2</b>	67%	33%
	<b>EN1</b>	12%	88%

Three levels of accumulated knowledge were identified among the subjects: Little, Some, and Significant. Table 5.6 organizes the relative frequency data by these groups. Additional detail can be found in Appendix B, including charts that depict graphically the percentage numbers reported in the table.

Three of the four subjects with *little* accumulated knowledge spent two thirds of their actions in the program and situation models. Three of the four subjects with *some* prior experience spent over 80% of their actions in the program and situation models. Two of the three programmers with *significant* accumulated knowledge spent one third of their actions in the program and situation models.

We did not see a trend of decreasing program and situation model references as accumulated knowledge increased. There appear to be trends within each category related to task or strategy. For example, the three subjects who had *little* prior experience with the software and who had two-thirds of their actions in the program and situation models were either working on debugging tasks or an enhancement that included an element of debugging. The other subject **G1** in that category used an opportunistic strategy during code understanding that necessarily kept his focus at a high level of abstraction rather than at the program model level.

We suspect that accumulated knowledge is important, but that task and expertise may be more important. One way to see how accumulated knowledge changes the un-

derstanding process is through a controlled experiment in which both task and expertise are held constant and accumulated knowledge is varied among a richer and more precisely defined set of accumulated knowledge classifications.

## 5.2 Processes – Levels 2 & 1

Comprehension processes are the dynamic aspects of code understanding. Recall that processes support construction of top-down, situation, and program models and are a single level of abstraction below that of the model components. (See Table 3.1.) In this section we report on three types of processes: *systematic*, *opportunistic*, and *combined systematic/opportunistic*. These processes are based on process analysis of three subjects, **G2**, **AD1**, and **EN2**. Systematic understanding consists of straightforward sequences of actions that are repeated during the programming session. These sequences are driven by flow of control in program code. In opportunistic understanding we find triads of goals, hypotheses, and actions based on “a need to know” approach rather than flow of control. The combined systematic/opportunistic process falls somewhere between the first two, sometimes following control flow and sometimes opportunistically generating hypotheses and taking actions to resolve them.

### 5.2.1 Systematic Processes

Systematic processes are dynamic code understanding activities consisting of *episodes*, *aggregate processes*, and *session level processes*. As discussed in subsection 3.5. these activities are found at different levels of detail and support the Integrated model. Here we illustrate the types of processes discovered in a single subject’s (G2) program understanding protocol.

#### 5.2.1.1 Episodes – Level 2

Episodes are sequences of actions of various types carried out to accomplish a goal. They exist at level two (See Table 3.1) since they support the systematic comprehension process. Episodes containing common actions with similar goals emerge as the lowest level processes. Each episode is specified by its action sequences. To illustrate, episodic process *P1 – Read Block in Sequence* starts with the overall goal of understanding a specific block

of code, e.g. "I'm going to read the description and see if it gives me some good clues as to what's going on." Some of the observed actions that support the original goal are: generating hypotheses while reading comments, chunking information, making note of interesting aspects, but postponing their investigation.

Figure 5.1 presents this process in graph form as a state machine to illustrate the basic form of episodes. Arcs indicate action types while states represent level of understanding.

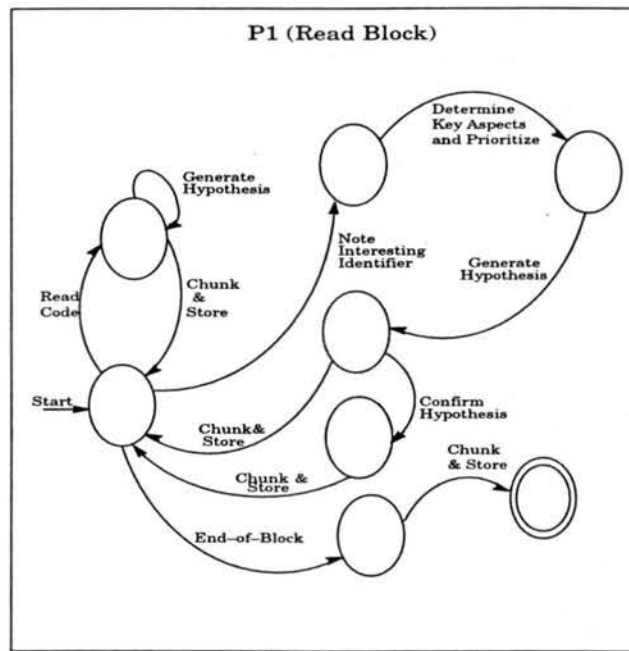


Figure 5.1: Systematic Processes: Episodic Process – Read Block

Two different action sequences characterized as a P1 episodic process illustrate different traces through the P1 state diagram of Figure 5.1.

1. Examine next module in sequence(Read Code) → Chunk & Store knowledge → Examine next module in sequence (Read Code) → Generate hypothesis → Chunk & Store.
  2. Read Code Comments → Generate Hypothesis → Chunk & Store → Note Interesting Identifiers & Determine Key aspects → Chunk & Store → Read next module in sequence → Note Interesting Identifiers & Determine Key aspects → Chunk & Store
- ...

Table 5.7 lists seven episodic processes and how often they occurred in the transcript (subject G2 (see Table 4.1)). It shows that the subject spent the majority of his time reading the code, determining the behavior of a variable, and incorporating this knowledge into his mental model of the program module. The engineer applied a *systematic strategy* of reading each line of code in approximate sequence.

Table 5.7: Episodic Process Frequency Count (Subject: G2)

<i>Episodic Process Name</i>	<i>Code</i>	<i>Number</i>
Read Block in Sequence	P1	7
Integrate Not Understood	P2	4
Determine Variable Def/Use	P3	7
Incorporate Acquired Program Knowledge	P4	5
Identify Block Boundaries	P5	2
Resolve Deferred Questions	P6	2
Understand a Procedure Call	P7	1

Table 5.7 also shows that preferred episodic processes exist. Processes P1 and P3 were the most frequent. These two processes are preferred during detailed understanding of one module. In section 5.4.2 (The Role of Code Size on Model Building) we report on an investigation of how the size of code to be understood might cause differences in the way programmers approach their task. Our results showed that as code size increases, the level of abstraction at which programmers tend to work also increases. In this situation, one would expect to see more episodes of type P2 (Integrate Not Understood) in understanding of larger software systems.

During our analysis we were able to associate information needs (and their frequencies) with action types and thus with episodic processes. Table 5.8 shows information needs for process P1 for subject G2. The three most frequently needed information types for P1 directly relate to the activities shown in the state diagram. E.g. determining the end-of-block condition requires code block boundary information.

Interestingly, we could not find episodic processes that occurred on a single level of abstraction and therefore classified as purely top-down, situation, or program model processes. Many episodes contained actions that were associated with all three integrated model components. This supports the idea that programmers constantly switch between model components (levels of abstraction). In this, episodic processes are similar to the

Table 5.8: Understanding One Module – Information Needs for Process P1

<i>Episodic Process</i>	<i>Information Needs</i>	<i>Number</i>
P1: Read Block	Code Block Boundaries	4
	Data Type definitions & location of identifiers	3
	Call Graph Display	2
	History of past modifications	1
	Data structure definitions tied to domain concepts	1
	Location of called procedures	1
	History of browsed locations	1
	Beacons tied to situation model or program model	1
	Description of system calls	1
	Location of documents for program & domain	1

design process reported in [24], which is not too surprising since cognition can be considered a recreation of the design task. These results also show that for cognition aids to be effective, tools or documentation must support work at all levels of abstraction and the frequent switches between them.

Individual episodes can vary greatly, because their goals are very different. An episode may use the same action types as another but occur in a different order. As we combine and abstract commonalities at different process levels, the processes themselves become similar. This corresponds to similar higher level goals which the aggregate and session level processes support. However, they use different episodes as their low level tactics. This is anticipated because Ericsson ([20], page 196) had a similar result in a study of subjects solving an Eight Block Puzzle. The solutions of individual subjects varied significantly when analyzed at the lowest level of detail. However, when the same solutions were analyzed at a more abstract level, the subjects followed more predictable sequences of actions.

#### 5.2.1.2 Aggregate Processes – Level 1

Similar to the role of action types in episodic processes, episodic processes are the components of aggregate processes around a common goal. Different aggregate processes use different sets of component episodic processes. Table 5.9 shows the composition of aggregate processes.

Three aggregate processes were discovered in G2's protocol and we illustrate one below in Figure 5.2 as a state diagram.

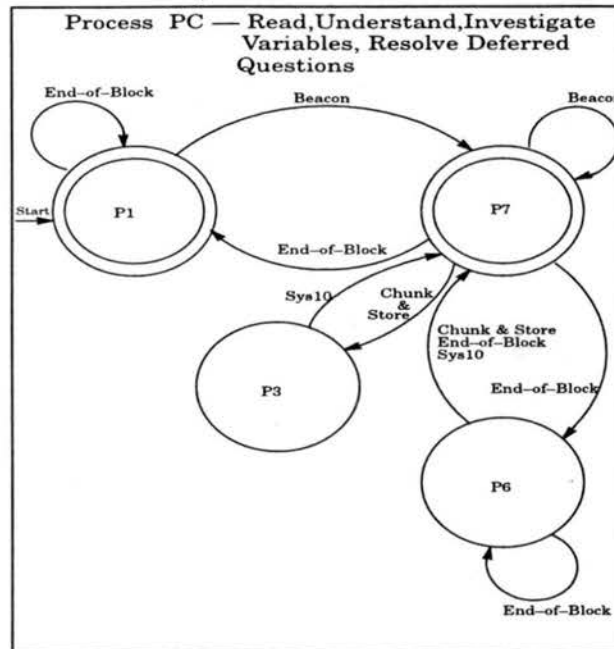


Figure 5.2: Systematic Processes: Process PC – Aggregate-Level

Table 5.9: Aggregate Processes – Episodic Composition

<i>Episodic Process</i>	<i>PA</i>	<i>PB</i>	<i>PC</i>
P1:Read Block in Sequence	X	X	X
P2:Integrate Not Understood	X		
P3:Determine Variable Def/Use	X	X	X
P4:Incorporate Acquired Program Knowledge		X	
P5:Identify Block Boundaries		X	
P6:Resolve Deferred Questions			X
P7:Understand a Procedure Call			X

Table 5.10 shows frequencies of aggregate level processes. At the aggregate level, processes PA, PB, and PC begin to look very similar. Each aggregate process contains a *Read* component and a *Investigate variables* component. *End-of-Block* and *Beacons* typically trigger the end of one episodic process and the beginning of another or the end of the aggregate process itself. (Figures C.12, C.13, and C.14 in Appendix C. show the state diagram for these aggregate processes.)

Table 5.10: Aggregate Process Frequency Count (Subject: G2)

<i>Aggregate Process Name</i>	<i>Code</i>	<i>Number</i>
Read, Integrate, Investigate variables	PA	4
Read, Incorporate acquired program knowledge, Investigate variables, Identify Block Boundaries	PB	3
Read, Understand, Investigate variables, Resolve deferred questions	PC	3

Table 5.11: Process Trigger Frequencies (Subject: G2)

<i>Process Trigger</i>	<i>Frequencies</i>	
	<i>PC</i>	<i>All Aggregate Procs</i>
Beacon	7	14
Chunk & Store Knowledge	2	8
End-of-Block	7	20
End-of-Stack	0	2
Understanding strategy determined (Sys10)	2	7

*Triggers* cause state changes between processes. They can be code induced (e.g. end of code block) or an action type (e.g chunk and store knowledge). Table 5.11 lists the triggers found in the example protocol and their frequencies for Process PC and the total for all the aggregate processes. Beacons and end-of-block triggers were the most frequent triggers. Again, this could be a by-product of the systematic strategy used by this subject. E.g., a jump out of episodic process P1 (Read Block in Sequence) into process P7 (Understand Procedure Call) occurs when G2 encounters an unrecognized procedure call (a beacon). He decides to understand what the procedure does. After investigating it he reaches its end (End-of-block) which triggers the end of P7 and resumption of P1 where he last left off.

### 5.2.1.3 Session Level Processes – Level 1

Session level processes are at the highest level. The state diagram in Figure 5.3 was derived in the same way as the aggregate-level processes by tracking the sequences of aggregate-level processes. This diagram represents a general understanding maintenance task. At the highest level, only “End-of-block” and “Chunk & Store” cause switches from one aggregate-level process to the next.



The session-level process (for Understanding a single module) shows that for this subject all the aggregate-level processes represent investigation towards building chunks [36, 45]. Chunking is an important abstraction mechanism in code understanding. Thus, at the session-level the purpose of each aggregate process is to understand a block of code (using different detail steps and information) and then to chunk and store the learned information.

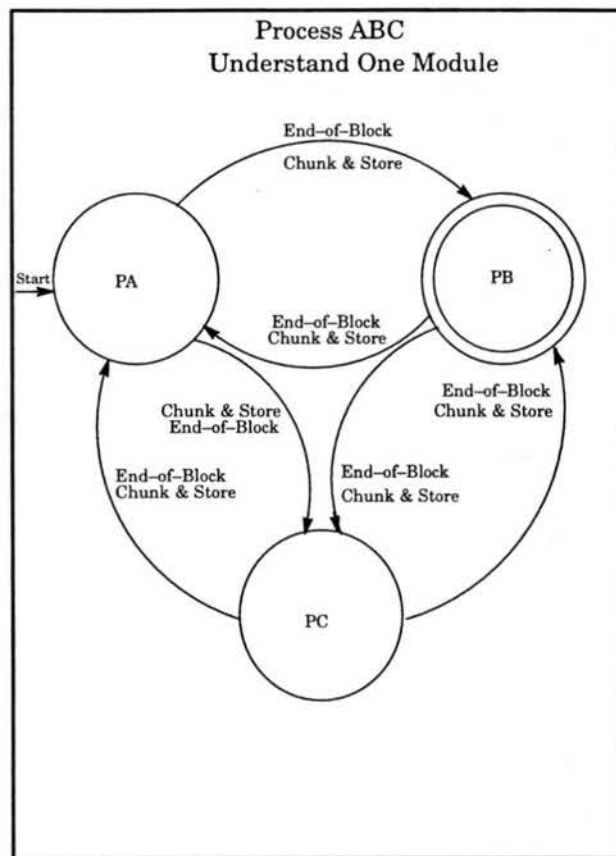


Figure 5.3: Systematic Processes: Process ABC – Session Level

#### 5.2.1.4 Interpretation of Systematic Processes

As we aggregate, the processes become similar but their contents at the episode-level are not. Below the episode level we find different sequences of activities which are themselves abstracted encodings of similar steps in the protocols.

Processes PA, PB, and PC begin to look very similar at the aggregate level. One conjecture is that these aggregate processes represent instances of a similar higher level

strategy. Subject G2, whom we used as an example to illustrate the process analysis, demonstrated a systematic approach [27] to understanding. Another common strategy used for code comprehension is opportunistic, one in which the programmer follows individually determined, *relevant* threads through code and documentation. For this type of strategy we did not see the nicely organized processes we saw with systematic understanding. We investigate this behavior next.

### 5.2.2 Opportunistic Processes – Level 1

Opportunistic understanding is very different from systematic understanding. Instead of following control-flow sequences dictated by the code, opportunistic processes involve understanding only those elements perceived as relevant to the current task. The main components we find during this type of process are goals, hypotheses, and actions directed at resolving these hypotheses. This *triad* of opportunistic understanding components are related. For example making a goal such as “determining what is wrong with a program” leads to generation of hypotheses such as “the cause of the buggy behavior”. Hypothesis generation leads to actions which help to confirm or refute the hypothesis.

We use this triad of goal, hypothesis, action as a way of explaining the opportunistic processes we saw in the protocol analysis of subject **AD1** who worked on an adaptation task. While the systematic processes discussed in the previous section were represented as state diagrams, the opportunistic processes are better represented as trees of triads.

#### 5.2.2.1 Hypothesis Resolution

Hypotheses are important drivers of cognition. They help to define the direction of further investigation. Generating hypotheses about code and investigating whether they hold or must be rejected is an important facet of code understanding.

There are three ways in which an individual hypothesis can be resolved. It can be abandoned, confirmed, or it can fail. **AD1** generated a total of 40 hypotheses during the programming session. Of 40 hypotheses, 17 were confirmed, 15 were abandoned, and 8 failed. This high number of abandoned hypotheses is surprising in light of Brooks [9] who found that programmers rarely abandon hypotheses. Rather, they backtrack to find a different way to confirm hypotheses. In our study, the subject did not backtrack when

abandoning an hypothesis. Additionally, there were a significant number of abandoned hypotheses. While this appears to contradict Brooks [9], there may be another explanation having to do with the expertise of the subject. Vessey [53] found that experts are flexible in their approaches to problem comprehension. They are able to let go of questionable hypotheses and assumptions more easily than novices. AD1 was an expert and thus expected to easily and quickly abandon questionable hypotheses. One could argue that the large number of (later abandoned) hypotheses speaks to the flexibility in which the subject approaches the comprehension problem. This appears to confirm Vessey's findings.

We distinguish between *conscious abandonment* and *forgetting* of hypotheses. Forgotten hypotheses are those to which the subject never returns. Half of the abandoned hypotheses were forgotten and half were consciously abandoned. Consciously abandoned hypotheses were explicitly rejected for three reasons: they were complex and the effort it would take to confirm was not worth the return; they became irrelevant; a more efficient approach to meet the goal was available. Sometimes the hypotheses were forgotten because another task proved more fruitful. In other cases, it was not possible to determine why the hypothesis was forgotten.

Only 20% of the hypotheses failed. Six out of eight hypotheses failed right away (within seconds) while the remaining two took on the order of several minutes to fail. One of the latter two involved a simple hypothesis. The problem was that the subject needed a tool. This tool took a great deal of time to report an answer that would support or reject the hypothesis. Because the majority of the hypotheses that failed took so little time to fail, we suspect that hypotheses that are easy to resolve are pursued more often than those that are time consuming or may not be worth the effort. This may explain the relatively few number of time consuming failures and relatively large number of abandoned hypotheses.

Thus, the (expert) judgment of effort and pay back related to an hypothesis may explain why experts abandon hypotheses and novices do not. Novices cannot make that judgment.

### 5.2.2.2 Dynamic Resolution Process

Next, we analyzed relationships between hypotheses, how sequences of goals and hypotheses affect achieving the overall goal of the task, and tried to identify an overall process of hypothesis generation and resolution.

The protocol contained 20 major goals. For each goal, the analysis excerpted associated hypotheses, supporting comprehension actions, and any subsidiary goals, hypotheses, and actions. The result of the analysis is represented graphically. Each figure shows analysis results for an individual top-level goal. Ellipses represent goals, rectangles are hypotheses, and actions are triangles. Arrows show the flow of goal/hypothesis resolution. The flow of goals, hypotheses, and actions over time is determined by a left to right depth-first traversal of the tree. Each figure is accompanied by descriptions of each goal and supporting subgoals, hypotheses, and actions.

Figures 5.4(Goals 1 & 2), 5.5(Goal 4), and 5.6(Goal 7) show a representative subset of the 20 goals. They range from simple (Figure 5.4) to complex (Figure 5.6). Goal 1 represents a simple goal with an associated hypothesis which is immediately abandoned. Goal 2 leads to an hypothesis which is evaluated progressively, first by memory, then with a tool; a subsidiary hypothesis is abandoned leading to the failure of the hypothesis. The following text represents a shortened version of the protocol to illustrate the thought process for Goals 1 and 2.

#### GOAL 1 (Figure 5.4)

What he was doing: Invoked the program to watch it run incorrectly.

Goal 1: Determine what was wrong by watching program behavior.

**Hypothesis 1:** Cause of the buggy behavior and generation of alternative hypotheses. This hypothesis was never returned to and is considered abandoned.

#### GOAL 2 (Figure 5.4)

What he was doing: Looked at an Imake file.

Goal 2: To determine what libraries are needed by the program.

**Hypothesis 2:** He guessed at how many; then listed them by name from memory. He tried to confirm this by using a tool that listed the libraries, including libraries used by other libraries.

*Sub-Hypothesis 2.1:* Result of tool output led to acknowledging his original hypothesis (#2) was incorrect; there

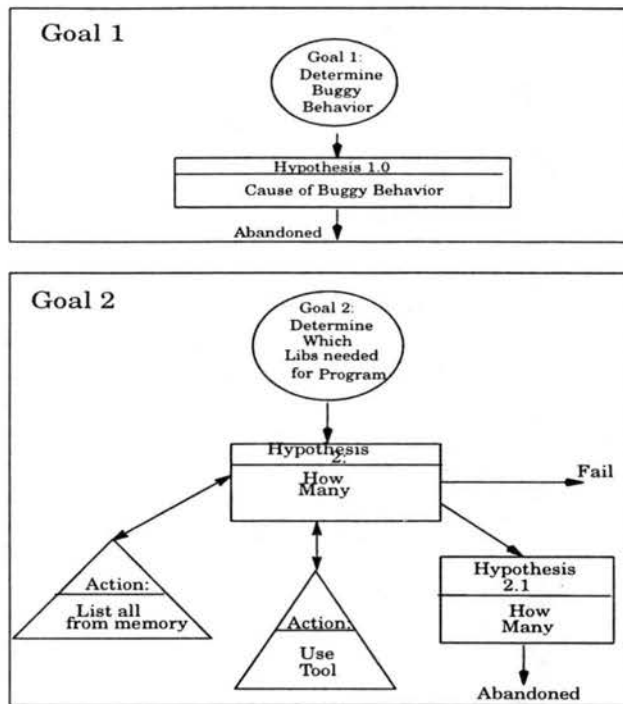


Figure 5.4: Opportunistic Processes: Goals 1 and 2

were more than his original estimate. He hypothesized that there were at least 4 libraries involved, but immediately abandoned this hypothesis because of the amount of work it would take to confirm it. He judged the answer to be irrelevant to the overall goal of porting the program.

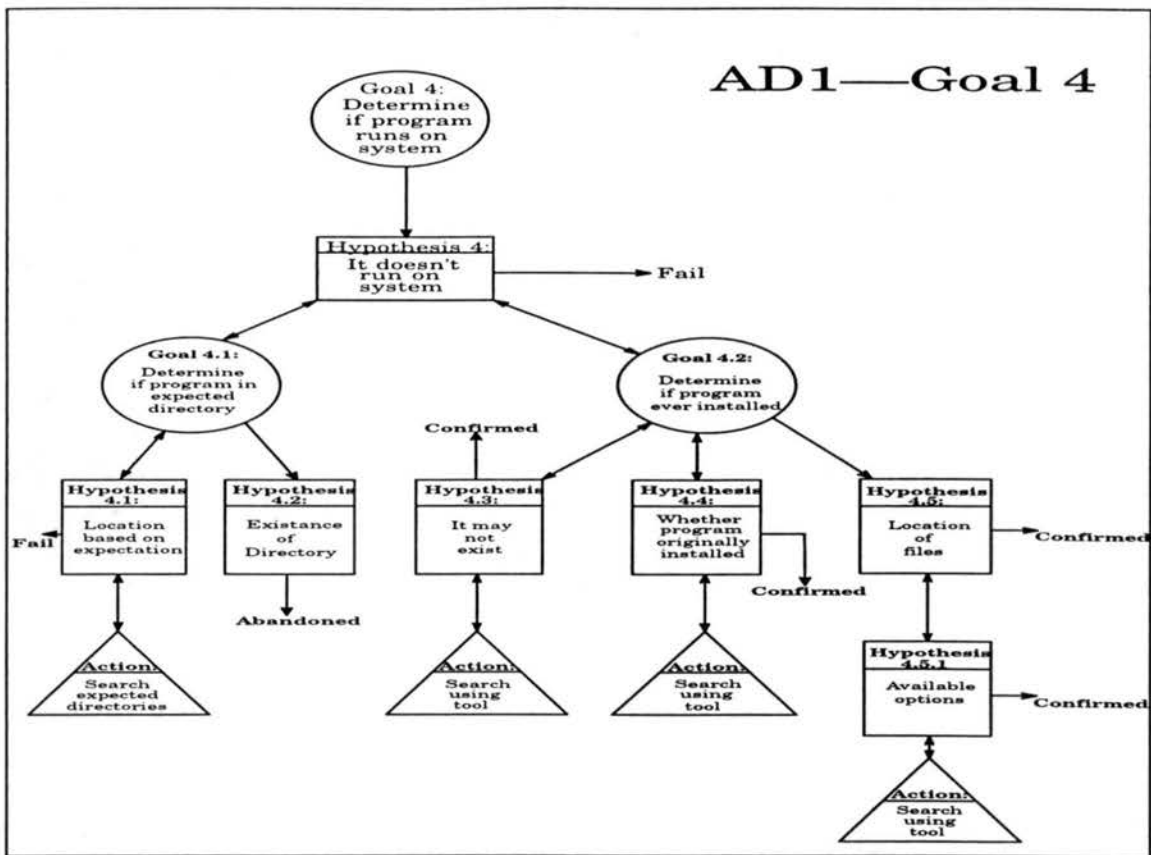


Figure 5.5: Opportunistic Processes: Goal 4

The following text represents a shortened version of the protocol to illustrate the thought process for Goal 4.

GOAL 4 (Figure 5.5)

What he was doing: See if the program runs on a particular system. He had never seen it on that platform.

Goal 4: Get it to run on his system if it was there.

**Hypothesis 4:** Doesn't exist on that system. In order to determine if it was there or not he generated two sub-goals.

*Sub-Goal 4.1:* Determine if the program is where he expected.

**Sub-Hypothesis 4.1:** He guessed at the location. It failed after a search of the directory he expected it to be in. This led directly to the next sub-hypothesis.

**Sub-Hypothesis 4.2:** Hypothesis about the existence of another directory. Hypothesis was abandoned in favor of alternative goal/hypothesis sequence #4.2 instead of searching for the directory itself. The alternative directory was never mentioned again.

*Sub-Goal 4.2:* Determine if the program was ever installed as part of the operating system.

**Sub-Hypothesis 4.3:** Program doesn't exist on his system (space limitations). He invoked a system-wide *find*

command from a particular place within the file system and then went off to other things while the `find` was executing. Eventually, the `find` returned with nothing found. Thus, his hypothesis was confirmed; the program did not exist on that part of his system.

**Sub-Hypothesis 4.4:** He then decided that he did not use the `find` command from the correct place. He reasoned that if he went to the root directory on his system and issued a `find` from there, he would be able to tell if the program had ever been installed. Again, he issued the `find` command and went on with other things. The `find` command returned with nothing found, confirming his hypothesis that the program was never installed.

**Sub-Hypothesis 4.5:** At this point he knew the program was never installed but suspects the program is in the release of the operating system somewhere. He hypothesized it was somewhere in the OS release bits. This led to another sub-hypothesis about the correct `find` command to issue.

*Sub-Hypothesis 4.5.1: He thinks there are options to tell the `find` command program to search case insensitive. He looks for the option by reading man pages on the computer. This hypothesis is confirmed when he finds the correct option.*

He then uses the `find` command with the appropriate options from within the O/S directory. The tool finds the files. His top-level hypothesis (#4) that the program does not exist, fails.

Goal 4 is obviously more complicated. Its supporting hypothesis leads to two subsidiary goals, representing a progression of knowledge. Each of these subsidiary goals in turn leads to second level hypotheses. They also reflect a progression of analysis and acquired knowledge. Actions support the necessary fact-finding. Goal 4 is an example of an arsenal of skills (hypotheses and actions) to be applied to a problem (in this case to determine whether the program runs on the system).

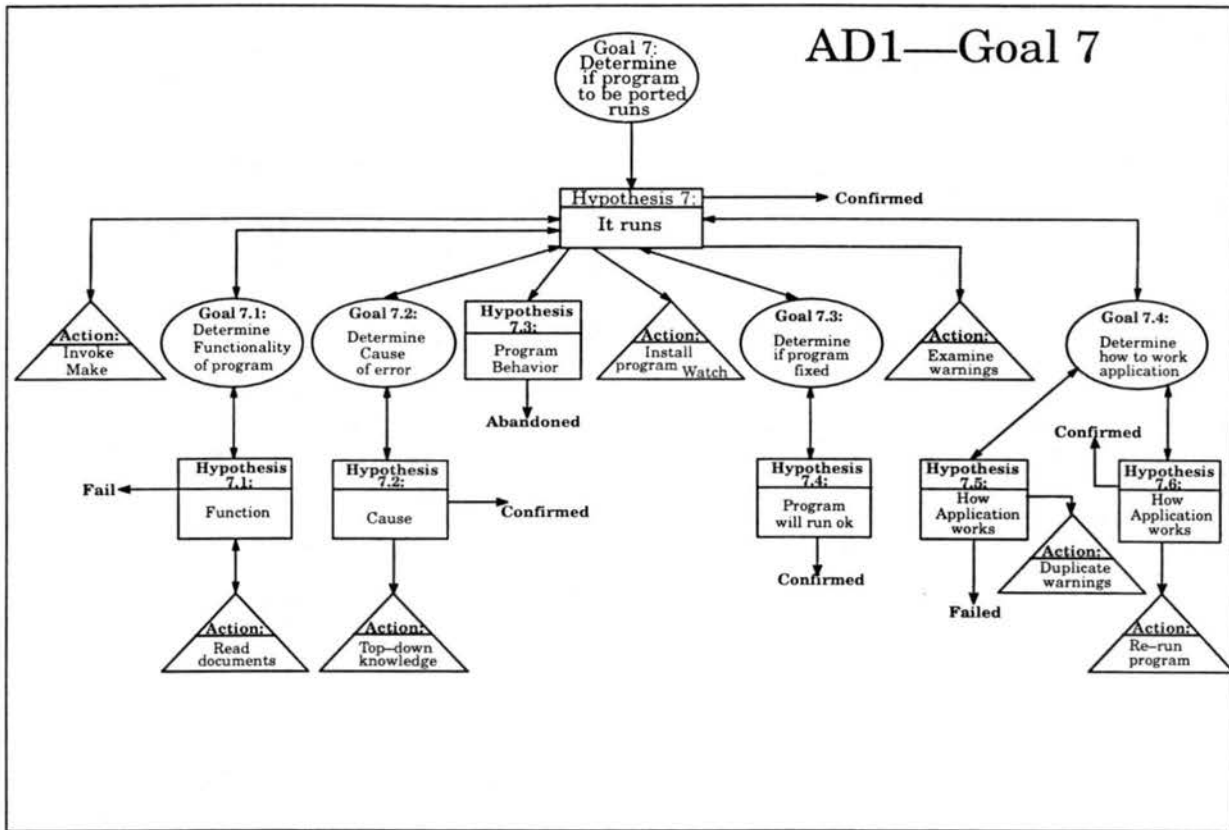


Figure 5.6: Opportunistic Processes: Goal 7

The following text represents a shortened version of the protocol to illustrate the thought process for Goal 7.

GOAL 7 (Figure 5.6)

What he was doing: Try to port another client.

Goal 7: Determine if the program runs.

**Hypothesis 7:** He hypothesized that the program runs because it had successfully compiled at some point prior to the programming session. He reasoned that he could re-build the program by executing a *Make* command and then invoke the newly built client to watch it run. Several subgoals, hypotheses, and actions followed that confirmed the program was running as expected.

**Goal 7.1:** Determine the program's functionality.

**Hypothesis 7.1:** He hypothesized that this program was a ping-pong game. He brought up the associated man pages and determined that instead it was a physical simulation of an ideal gas in a heated box.

**Goal 7.2:** He noticed the program was not behaving as explained in the *man* pages. He decided to determine the cause.



**Hypothesis 7.2:** Hypothesized the cause as related to warnings about missing files. He used top-down knowledge about client architecture to confirm this hypothesis.

*Hypothesis 7.3:* He looked at the *Make* file and hypothesized that the missing files would be overwritten during the build process. This hypothesis was never returned to and considered abandoned.

*Action:* He installed the program and then invoked it so he could watch it run. During the install process, he returned to Goal 4 before coming back to this one.

*Goal 7.3:* After installing the program he wanted to confirm that it could find the missing files by executing the program.

**Hypothesis 7.4:** Hypothesized that the program will run this time, that the missing files will be found. This hypothesis was immediately confirmed (no warning messages).

*Action:* The program was run, new warning messages were examined.

*Goal 7.4:* To understand the new warning messages he decided to determine how to use the application.

**Hypothesis 7.5:** Hypothesized about how the application works. He was convinced he had to use a trick to get the program to function correctly. He tried to duplicate the warnings by re-running the application but this time did not get the warnings. This hypothesis failed when he realized he was missing some required input.

**Hypothesis 7.6:** Based on the results of the previous hypothesis, he hypothesized that the program worked differently. He tested his theory and this time the program behaved as expected.

As with Goal 4, the programmer applied an arsenal of skills to his goal. This accounts for the relatively shallow tree.

### 5.2.2.3 Interpretation of Opportunistic Processes

Goal 1 (Figure 5.4) is representative of simple goals. A goal is stated and leads to an hypothesis which is immediately abandoned. Goal 2 (Figure 5.4) is more complicated, but still relatively simple: a goal is stated, an hypothesis is generated. It is supported by 2 actions and a sub-hypothesis. Goal 4 (Figure 5.5) includes 2 sub-goal trees supported by additional hypotheses and actions. Finally, Goal 7 (Figure 5.6) is the most complicated of the four. The major hypothesis generates a wide, shallow tree consisting of single actions and sub-goal trees.

A majority of the goals were *sequential* in that once stated they were pursued until closure was reached. Goal 4 and Goal 7 are the exception. Goal 4 was *interleaved* between

three other goals during the session. Goal 7 was briefly interrupted to concentrate on Goal 4. Goal 4 represents a tree structure with symmetric subtrees. This, we believe, is because each sub-goal and hypothesis were similar, i.e. the subject was trying to find the location of the program and if it was originally installed on his system. The actions supporting the hypothesis involve an arsenal of search mechanisms, typically a software tool. Each of the hypotheses and actions were simple enough to interrupt (for example, during the time required for the search tool to return a result) and go on to another goal. Using this arsenal of techniques for resolving hypotheses, our subject was able to easily return after finishing another goal and go on with the next step in completing goal 4.

In contrast, goal 7 is much more complicated. This, we conjecture, results in the fairly wide and shallow tree structure of Figure 5.6. Each of the sub-goals, hypotheses, and actions are complex enough that interruption could cause the goal or hypothesis to be forgotten. Goal 7 was interrupted during installation of the client program. In this case, cognitive load was not overwhelming, the engineer was able to return to goal 4.

A fairly obvious process falls out of the analysis of these goals. In the simple cases we see goals resolved using a common pattern: goal-hypothesis-action. Goal 2 (Figure 5.4) contains two of these triads. In more complex cases, we see the triad as part of subgoals and hypotheses. Goal 4 (Figure 5.5) has three, while goal 7 includes five. If we were to consider abandonment, failure, and confirmation of hypotheses as actions, there would be many more such triads. The overall dynamic process consists mainly of these triads applied over time. This straight forward process is interesting because, although our subject took an opportunistic approach to understanding, he proceeded in a straightforward, methodical manner consisting of sequences of these triads with a small amount of interleaving.

#### **5.2.2.4 Overall Comprehension Process**

Figure 5.7 shows the sequence of all 20 goals. Circles represent the goals and arrows indicate sequence. A solid arrow between two goals means the goal was complete before the next goal was begun. A dashed arrow represents interleaving between goals. The figure shows an obvious understanding process. Goals are resolved sequentially unless interleaving of a fairly simple goal can be accomplished efficiently. The subject had experience with the domain and the task. He applied an arsenal of techniques to resolving the

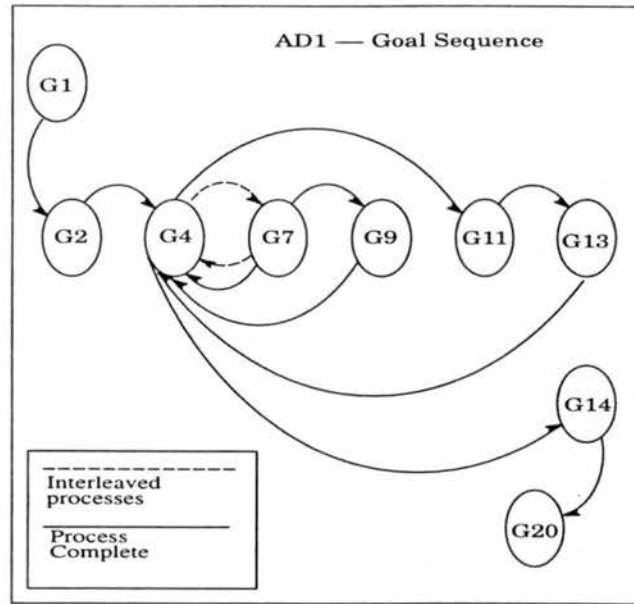


Figure 5.7: Opportunistic Processes: Goal Completion

goals he generated. If he perceived an hypothesis or goal to be too difficult to resolve, he usually abandoned it in favor of a more efficient alternative. If the return on investment of resolving a hypothesis was low, the hypothesis or goal was abandoned. If the goal or hypothesis seemed relatively easy to resolve, he usually pursued it as long as it was efficient and as long as it seemed the results would be worth the effort.

Goals and hypotheses are important facets of program understanding. Understanding concentrated on system level aspects (file structures, libraries, Makefiles, etc.). The subject was an expert in the domain and had prior experience with the code and the porting task, as he had ported similar programs before.

While our results are based on the analysis of a single subject's protocol, several interesting results can be used as the basis of more controlled experiments:

1. Goal/hypotheses resolution is a dynamic process consisting of sequences of goal-hypothesis-action triads. Interleaving of goals is rare and occurs when the cognitive load of doing so is not overwhelming.
2. Experts pursue hypotheses and goals that are easy to resolve and tend to abandon those that are time consuming with little return on investment, confirming Vessey's ear-

lier research results [53].

3. Experts have an arsenal of possible hypotheses and supporting actions they employ to understand code. This represents itself in the series of slightly modified hypotheses associated with goals (e.g. Goal 4)

### 5.2.3 Combined Systematic & Opportunistic Processes

A third comprehension process was discovered during the analysis of subject EN2's protocol which falls somewhere between the opportunistic and systematic processes. EN2's task was to add an enhancement which consisted of providing safety check functionality. (Previously the code did not check for "disk full" under certain conditions when a new file needed to be saved to disk. EN2 added the safety check as an enhancement.)

Because EN2's process was sometimes systematic and sometimes opportunistic, it was difficult to pictorially represent his processes using either of the graphical methods used in the previous processes subsections. Instead, we present a hierarchical list of high-level episodes and related sub-episodes.

#### 5.2.3.1 Systematic/Opportunistic Process Episodes

The following hierarchical list of episodes illustrates the combination of systematic and opportunistic understanding.

- E1: Set up environment, determine task approach.
- E2: Step through program as it executes until reaching a point where enhancement should go. Determine code to be examined more carefully.
- E3: Determine where to put error message for safety check.
  - E3.1 - E3.3: Step through code, generate hypothesis, determine message does not go there.
  - E3.4: Hypothesize location to put message.
  - E3.5: Determine where to put code and what to implement.
    - \* E3.5.1: Read block of code, determine where to put message.
    - \* E3.5.2: Determine if other procedures call routine to be changed.
      - E3.5.2.1: Propose solution
      - E3.5.2.2 - E3.5.2.4: Examine procedure to determine if proposed change will affect it. (Includes generating hypotheses about whether each examined location will or will not be affected.)
      - E3.5.2.5: Determine if rest of modules are similar and do not need detailed understanding.

- \* E3.5.3: Investigate specific solution.
  - E3.5.3.1: Determine strategy for finding information, eliminate alternatives.
  - E3.5.3.2: Determine if change to module code at this level will produce desired behavior.
    - E3.5.3.2.1: Examine Data structure and associated code, understand functionality.
    - E3.5.3.2.2: Determine which code modifies data structure.
    - E3.5.3.2.3: Examine procedure, understand.
    - E3.5.3.2.4: Chunk and store what has been learned up to this point.
    - E3.5.3.2.5: Determine calling structure.
    - E3.5.3.2.6: Eliminate/refine possible solution.
    - E3.5.3.2.7: Study code, including eliminating possible solutions, searching for similar, existing code.
    - E3.5.3.2.8: Eliminate/Refine possible solution
    - E3.5.3.2.9: Determine solution, i.e. code change to specific module.
- E4: Make changes to code
  - E4.1: Set up environment
  - E4.2: Add Code
  - E4.3: Determine next task
  - E4.4: Add Code

**EN2** has several systematic/opportunistic episodes including E3.1 through E3.3 in which he single steps through the code (using a debugger tool) while he generates hypotheses and eliminates possible locations for the enhancement. Episodes E3.5.2.1 through E3.5.2.4 also include both systematic examining code and making hypotheses about whether the proposed change will impact code at each location he considered. **EN2** is never completely systematic, nor completely opportunistic.

#### 5.2.4 Interpretation and Conclusions on Comprehension Processes

We identified three comprehension processes: systematic, opportunistic, and systematic/opportunistic. In systematic processes we see well-defined sequences of actions repeated during understanding. These sequences can be aggregated until an overall process of code comprehension for specific maintenance tasks is identified. Opportunistic

processes consist of triad sequences of goals-hypotheses-actions. They are opportunistic in that hypotheses are resolved based on perceived relevance. Systematic/opportunistic processes include a combination of systematic and opportunistic understanding.

While we expect to see elements of each of these processes during program understanding, we did not see enough similarities across subjects to warrant further investigation. It appears that processes are dependent on individual strategies and preferences.

### 5.3 Hypotheses – Level 1

In section 5.2.2, we investigated the role of hypotheses in opportunistic understanding. Here we consider the types of hypotheses programmers generate, the role they play in switching between levels of abstraction, and whether task, expertise, or the amount of prior experience with the code impacts the number and types of hypotheses generated.

Switching between models can occur as the result of an hypotheses. For example, an hypothesis such as “I bet they use a bubble sort?” (a situation model hypothesis) may be followed by reading code comments (a program model action) to find the code implementing the sort. The hypothesis generated in the situation model was followed by a switch to a different model. We were not only interested in whether task, expertise, or accumulated knowledge influenced how often hypotheses generated a switch to a different model but also which models were most frequently the source or target of a switch in connection with an hypothesis.

We also wanted to investigate whether or how often comprehension according to Brooks’ model occurred among the subjects. This model of successively refining hypothesis from higher levels of abstraction to lower levels until actual code is reached for hypothesis verification occurs in only one direction: top-down.

When comparing the number of hypotheses to the number of knowledge references, it appeared that individuals with more expertise tended to make more knowledge references per hypothesis than less expert subjects. We developed a potential indicator of expertise, the *HK-Ratio*, which looks at the relationship between hypotheses and references to knowledge.

This section presents the results of the hypothesis analysis for all subjects. First, totals for types of hypothesis within each model for all subjects is given. The major results

on hypothesis types for each model grouped according to task, expertise, and accumulated knowledge follow. Hypothesis generated switching results are then presented. Finally, we present the HK-Ratio analysis and results.

### 5.3.1 Hypothesis Type: Overall Frequencies & Model Frequencies by Task, Expertise, & Accumulated Knowledge

#### 5.3.1.1 Hypothesis Type: Overall Frequencies

What are the types of hypotheses programmers make? Do they occur at all levels of the integrated model? Are there any hypotheses that are made more frequently than others? To answer these questions consider Table 5.12. It contains a description of each hypothesis type occurring in the eleven protocols. The *tag* column gives each hypothesis type a unique identifier. Hypotheses are classified as Top-Down, Situation, and Program model actions. Table 5.12 provides frequencies and relative frequencies for each hypothesis type (most frequent first).

For the top-down model, the most frequent type of hypothesis is about functionality (OPH1). This is consistent with results on action-types (section 5.1.1) and information needs (see section 5.6 below). The second most frequent type of action associated with the top-down model was generation of hypotheses about functionality. The most frequent type of information need concerned the need for domain concepts embodied in the program. Location/Status/Description/cause of error (OPH18) is the second most frequently found type of hypothesis for top-down model construction because of the preponderance of subjects working on debugging tasks (4 out of 11). This supports a conjecture that corrective maintenance requires a high number of hypotheses as a strategy for tracking down defects. Generating hypotheses regarding level and structure of the code (OPH16) is the third most frequent type of hypothesis. This is consistent with the results on action type analysis because “Determining next program segment to examine” is the sixth (out of 21) most frequent action found in the top-down model. During the programming sessions, trying to decide what was important to understand was a common goal. Frequently, this was associated with generating hypotheses about code structure in order to focus attention.

For program model hypotheses, the most frequent type concerned code correctness and causes and locations of errors (SYS16). Again, we believe this is due to the number of corrective maintenance tasks. The second most frequent type of hypothesis, statement execution order/state (SYSH6), compares well to the program model action references where the second most common action is to examine the next module in sequence. Hypotheses about program state are natural when one is traversing the code sequentially. Hypotheses about variable values and structure (third and fourth most frequent hypothesis type) represents the need to understand data and data flow. This fits well with Pennington's [36] explanation of the Program Model.

For situation model hypotheses, functionality hypotheses have the highest frequency. The connection between hypotheses and other action types is not obvious here. However, this parallels hypotheses about function at the top-down Model level. We can conclude that the reason situation model hypotheses about functionality are so frequent is because making connections between top-down and situation model were the focus of these hypotheses. This conclusion will be further supported when looking at hypotheses switching behavior (Table 5.16). Since the most common type of top-down hypothesis concerns functionality, it follows that situation model hypotheses should be about functionality if the main purpose is to draw the connection between the top-down and situation models.



Table 5.12: Hypothesis-Type Frequencies – All Tasks

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	65	27%
	OPH18	Location/Status/description/cause of error	34	14%
	OPH16	Level & structure of code/scope	28	12%
	OPH9	Permissions/Environment set correctly/ Tool functionality	16	6%
	OPH8	Program functions correctly	14	6%
	OPH14	Available functionality	14	6%
	OPH13	Number/type/location of file	10	4%
	OPH5	Existence of installed (running) program	8	3%
	OPH2	Variable functionality/domain concepts	7	3%
	OPH3	Rules of discourse/Expectations	7	3%
	OPH6	Existence of specific functionality	7	3%
	OPH7	Number/type/existence/location of libraries	7	3%
	OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	6	2%
	OPH4	I/O behavior	5	2%
	OPH10	Location to add functionality	4	2%
	OPH11	Comparison of functionality at high level	4	2%
	OPH17	Design Decisions/Modifications	4	2%
	OPH15	Approaches/relative difficulty in making change	1	0%
	OPH19	Current location	1	0%
Program Model	SYSH16	Code correctness, cause/location of error	29	17%
	SYSH6	Statement execution order/state	25	14%
	SYSH7	Variable value/defaults	22	13%
	SYSH4	Variable structure	17	10%
	SYSH2	Function/procedure function	15	8%
	SYSH1	Variable function	14	8%
	SYSH10	Syntax meaning	9	5%
	SYSH18	Location to add code/alternatives	7	4%
	SYSH5	Location/type/existence of function call	5	3%
	SYSH14	Code block function	5	3%
	SYSH8	(Non-)Existence of construct (var/code)	4	2%
	SYSH13	Code block/procedure comparison	4	2%
	SYSH19	Code block boundary location	4	2%
	SYSH20	Params/type definitions in procedure call	4	2%
	SYSH11	Design decisions	3	2%
	SYSH9	Variable/construct equivalency	2	1%
	SYSH12	Variable definition & it's location	2	1%
	SYSH17	Changes made correctly	2	1%
	SYSH3	Procedure/function call behavior	1	1%
SYSH15	Relevance of error in Makefile	1	1%	
Situation Model	SITH3	Function/procedure function, call function	44	31%
	SITH7	Existence of functionality/algorithm/variable	24	17%
	SITH1	Variable function	23	16%
	SITH2	Function/code block execution order/state	18	13%
	SITH5	Cause of buggy behavior	12	8%
	SITH8	Program function	12	8%
	SITH4	Effect of running program	6	4%
	SITH6	Comparison of terms/acronyms /functionality	5	3%
	SITH9	Design Decisions	0	0%

### 5.3.1.2 Hypotheses Types – By Task

One of the questions we are interested in answering is whether the types of hypothesis we see within each model are different for different tasks. The tasks are adaptive maintenance, corrective maintenance, enhancement, and general understanding. Appendix B (section B.2.1) contains a separate table for top-down (Table B.3), program (Table B.4), and situation model (Table B.5) hypotheses organized by task. These tables contain the absolute and relative frequencies of each type of hypothesis for each group of subjects working on a particular task. Additional analysis is also provided in the appendix. The main results are summarized in this section.

For corrective maintenance, hypothesis generation may be predictable in that at all levels (top-down, program, and situation models), hypotheses about code correctness dominate the type we see (OPH18 -location/status/description/cause of error; SYSH16, code correctness, cause/location of error;). The same appears to be true about adaptive maintenance, but here the influence may be more related to their specific task and where these subjects were focused at the time of the programming session. Not much can be said about enhancement because of the low number of hypotheses generated at each level. Finally, for general understanding, strategy (systematic vs. opportunistic) seems to be the dominant factor. Most hypotheses generated in the program model were made by **G2** (systematic understanding), while most of the top-down hypotheses were generated by **G1** (opportunistic understanding).

We were not able to conclude much about the types of hypotheses programmers make while working on a particular maintenance task. However, we may be able to discern patterns in the number of hypotheses generated within each model. Questions here concern how the number of hypotheses in the program, situation, and top-down models vary depending on the task and whether the variations are particular to each maintenance task. Table 5.13 contains the total count and relative frequencies of hypotheses for each task type. It also has total counts on a per subject basis within task groups for each model.

Table 5.13 shows that adaptive maintenance has very few program model hypotheses and the majority of the hypotheses were at the top-down level. This is very close to the behavior we saw for these subjects for action types (see Table 5.4): the adaptive

Table 5.13: Task Hypotheses by Model-Total & Per Subject Counts with Total Frequencies

<i>Model</i>	<i>Model Counts &amp; Frequencies</i>	<i>Adaptive Maint. 2 Subjects</i>	<i>Corrective Maint. 4 Subjects</i>	<i>Enhancement 2 Subjects</i>	<i>General Understanding 3 Subjects</i>
<i>Top-Down</i>	Total	65	77	18	82
	Frequency	71%	37%	28%	41%
<i>Situation</i>	Total	20	64	5	55
	Frequency	22%	31%	8%	27%
<i>Program</i>	Total	6	65	41	63
	Frequency	7%	32%	64%	32%
<i>All</i>	Ttl Hypotheses	91	206	64	200
<i>Top-Down</i>	Ttl Per Subject	32	19	9	27
<i>Situation</i>	Ttl Per Subject	10	16	2.5	18
<i>Program</i>	Ttl Per Subject	3	16	20.5	21
<i>Total</i>	Per Subject Hypoth.	45	51	32.0	67

maintenance subjects had 67% of their actions in the top-down model, 14% in the situation model, and 19% in the program model. Hypotheses generated by the corrective maintenance subjects were fairly evenly distributed across all three models. Again, this is very similar to the behavior we saw for this group's action types (Top-Down Actions = 33%; Situation Model Actions = 28%; Program Model Actions = 39%). Enhancement hypotheses occurred mainly at the program model level, which also matches the action-type frequencies. However, General Understanding hypothesis generation does not follow the same pattern. General Understanding hypotheses occur slightly more often at the top-down level while actions occur slightly more often at the program model level. For understanding, strategy (systematic vs. opportunistic) seems to be the dominant factor. The subject who took the systematic approach generated significantly more hypotheses at the program model level while the opportunistic subject generated mostly top-down hypotheses. Not much has been said about the third subject, L1, in the General Understanding group. This is because he had roughly the same number of program, situation, and top-down model hypotheses, with a slight preference for program model. The types of hypotheses within each model were very similar to the subject who applied systematic understanding. We believe that (L1's) lack of domain knowledge would tend to tie him to hypotheses about program construct functionality and statement execution order.

Looking at the *Per Subject* data in Table 5.13 we can see that overall, general understanding needs the most hypotheses and enhancement needs the least.

### 5.3.1.3 Hypotheses Types By Expertise

The next question to ask is whether expertise affects the types and frequencies of making hypotheses. We distinguish between three types of expertise: Language Only, Domain Only, and Language-and-Domain. Appendix B, section B.2.1.2 contains a separate table for top-down (Table B.6), program (Table B.7), and situation model (Table B.8) hypotheses organized by expertise level. These tables contain the absolute and relative frequencies of each type of hypothesis for each group of experts. Additional analysis is also provided in the appendix. This section summarizes the main results.

Conclusions about the role of expertise in making hypotheses are spotty. For top-down hypotheses, very few were made by language-only experts (24 hypotheses), the domain-only experts had hypotheses related to debugging tasks (OPH8,OPH12,OPH18) 40% of the time, indicating that these hypotheses were task rather than expertise related. The top-down hypotheses generated by language-and-domain experts were generated 84% of the time by a single subject.

In the program model, language-only experts generate hypotheses about program control-flow (SYSH6) and syntax meaning (SYSH10) reflecting Pennington's [36] conclusion that when code is unfamiliar, programmers tend to build control-flow views of the program first. Domain-only experts had mostly program model hypotheses about code correctness since three of four subjects were working on debugging tasks. No conclusions can be drawn for the language-and-domain experts since task seems to be the major influence for this group of subjects.

Language-only experts tend to make hypotheses that are useful for understanding control-flow (SITH1, variable function; SITH3, function/procedure function call function). These hypotheses support building a mental representation bottom-up by abstracting information at the program model level into the situation model level. Nothing could be concluded about hypothesis generation in the situation model for domain-only experts because they were mostly working on corrective maintenance tasks. Similarly, no conclusions could be made about domain-and-language experts as they used very dissimilar types of hypotheses for their tasks.

Next we would like to look at the total number of hypotheses within each model based on expertise. Questions here concern how the number of hypotheses in the program,

situation, and top-down models vary depending on expertise and whether the variations are particular to each type of expertise. Table 5.14 contains the total count and relative frequencies of hypotheses for each group of experts. It also has total counts on a per subject basis within the groups for each model.

Table 5.14: Expert Hypotheses by Model–Total & Per Subject Counts with Total Frequencies

<i>Model</i>	<i>Model Counts &amp; Frequencies</i>	<i>Language Only 2 Subjects</i>	<i>Domain Only 4 Subjects</i>	<i>Language &amp; Domain 5 Subjects</i>
<i>Top-Down</i>	Total	24	110	108
	Frequency	24%	53%	42%
<i>Situation</i>	Total	35	53	56
	Frequency	35%	26%	22%
<i>Program</i>	Total	40	44	91
	Frequency	41%	21%	36%
<i>All</i>	Ttl Hypotheses	99	207	255
<i>Top-Down</i>	Ttl Per Subject	12.0	28.0	21.6
<i>Situation</i>	Ttl Per Subject	17.5	13.0	11.2
<i>Program</i>	Ttl Per Subject	20.0	11.0	18.2
<i>Total</i>	Per Subject Hypoth.	49.5	52.0	51.0

From the table, we can say that the language only experts tend to generate mostly program and situation model hypotheses, domain experts generate more top-down model hypotheses than program and situation model hypotheses. Domain and language experts generate slightly more top-down references than program model references and the least amount of situation model hypotheses, not only among themselves but across all three groups. Thus, it appears that expertise influences which model experts tend to generate hypotheses in, but not necessarily which types of hypotheses they make.

The last row in Table 5.15 shows that there was very little difference in the total number of hypotheses generated by each group.

#### 5.3.1.4 Hypotheses Types By Amount Of Accumulated Knowledge

This section summarizes the main results about whether the amount of accumulated knowledge influences the making of hypotheses. Appendix B, section B.2.1.3 contains a separate table for top-down (Table B.9), program (Table B.10), and situation model (Table B.11) hypotheses organized by amount of accumulated knowledge. These tables contain the absolute and relative frequencies of each type of hypothesis for each group of

subjects with different amounts of prior knowledge of the software. Additional analysis is also provided in the appendix.

For top-down hypothesis generation, subjects with *little* prior accumulated knowledge about the code they maintain tended to generate a high percent of hypotheses about domain concept (OPH1) connections to known program functionality. Those with *some* accumulated knowledge tend to generate more of these types but at a lower rate than the first group. Those who have been working with the code the longest, not only generated this type of hypothesis at a lower rate than the previous two, it was also not the most common type. This trend appears to be related to amount of accumulated knowledge. Questions about connections of domain concepts to known program functionality are much more appropriate for people who are new to the code. If one has worked with the code for several years, these type of questions would have been answered long ago.

The subjects with *little* accumulated knowledge who generated the majority of the program model hypotheses were **C2**, **C3**, and **EN2**. All three subjects were working on finding and fixing defects, which may be why the second most frequent hypothesis (SYSH16) was about code correctness and location of defect (the most frequent was SYSH2, purpose of function). Because all were working on similar tasks and there were a small number of program model hypotheses generated by this group, it is difficult to draw conclusions about the effect of having *little* accumulated knowledge on generating program model hypotheses.

Hypotheses about the order of statement execution (SYSH6) was the most common type of hypothesis generated by those with *some* accumulated knowledge. However, the majority of these were generated by a single subject. Other common types of hypotheses generated by this group in the program model were about variable structure (SYSH4) and function (SYSH1). These subjects had seen only requirements and perhaps design documents or had done some work on the code already. Their control-flow view of the code was necessarily incomplete. The types of hypotheses they generated reflected their need to build the control flow view.

Accumulated knowledge can affect the types of situation model hypotheses generated. The most common type of hypotheses generated by those with *little* or *some* accumulated knowledge were about the purpose of a function or function call (SITH3). The goal of

Table 5.15: Accumulated Knowledge: Hypotheses by Model–Total & Per Subject Counts with Total Frequencies

<i>Model</i>	<i>Model Counts &amp; Frequencies</i>	<i>Little Acc. Knowledge 4 Subjects</i>	<i>Some Acc. Knowledge 4 Subjects</i>	<i>Significant &amp; Acc. Knowledge 3 Subjects</i>
<i>Top-Down</i>	Total	114	61	67
	Frequency	52%	28%	53%
<i>Situation</i>	Total	56	64	24
	Frequency	26%	30%	19%
<i>Program</i>	Total	48	91	36
	Frequency	22%	42%	28%
<i>All</i>	Ttl Hypotheses	218	216	127
<i>Top-Down</i>	Ttl Per Subject	28.5	15.3	22.3
<i>Situation</i>	Ttl Per Subject	14.0	16.0	8.0
<i>Program</i>	Ttl Per Subject	12.0	22.7	12.0
<i>Total</i>	Per Subject Hypoth.	54.5	54.0	42.3

forming connections between the program and domain models can be satisfied by using the situation model as a bridge between top-down and program model representations. Situation model hypotheses generated by those with *significant* accumulated knowledge were mainly about cause of buggy behavior (SITH5) and the effect of running the program (SITH4). This, we believe, is related to task since all but one of these hypotheses were generated by the adaptive maintenance subjects.

Obviously, these conjectures must be validated since our sample size is small. We need further experiments in which task and expertise is varied and accumulated knowledge is strictly controlled.

We are also interested in looking at the total number of hypotheses within each model based on amount of accumulated knowledge. Questions here concern how the number of hypotheses in the program, situation, and top-down models vary depending on the amount of prior experience with the code and whether the variations are particular to each group of programmers with *little*, *some*, and *significant* amounts of accumulated knowledge. Table 5.15 contains the total count and relative frequencies of hypotheses for each group of subjects. It also has total counts on a per subject basis within the groups for each model.

There are no discernible patterns within each model across the groups of subjects with differing amounts of accumulated knowledge. As discussed above, much of the behavior we saw could have been influenced by task. Overall, the table shows that there is a drop in

the number of hypotheses generated as the amount of accumulated knowledge increases. The difference between those with *little* accumulated knowledge and those with *some* is slight. The difference between those with *some* accumulated knowledge and those with *significant* prior experience is greater. We expect those with more experience to make more use of the knowledge they have acquired about the software.

### 5.3.2 Hypothesis Generated Switching Behavior

The integrated model of program comprehension assumes that programmers build mental representations of software at different levels of abstraction. A key part of this model encompasses the idea that these different levels are built essentially in parallel with each other. A method to determine this is to analyze how often focus on a particular model is changed to another. One way to change models is as the consequence of posing an hypothesis which requires actions at a different model level. We investigate this next.

#### 5.3.2.1 Hypotheses Generated Switching – All Subjects

Table 5.16 reports the number of switches caused by an hypothesis (for all subjects) related to a different comprehension model component. The row indicates the starting model and the column represents the ending model.

The table provides the total number of hypotheses in each model (column labeled *Number of Hypotheses*) to highlight the difference between the total number of hypotheses generated in each model and those that actually caused a switch. For example, there were 242 total top-down model hypotheses, but only 88 of these were associated with a switch to a different model (49 + 39). The total number of switches from a particular model to each of the other models is 266. This represents 46% of all hypotheses generated. That is, almost half the hypotheses result in model switches. We include the proportion of the 266 hypotheses for each type of switch to illustrate the distribution of hypothesis generated switches across models.

Table 5.16 shows that switching caused by hypotheses is pretty evenly distributed among the three models. This supports the theory that, in general, programmers switch between models and that hypotheses are important for building mental representations at different levels of abstraction. As we will shortly see, tendencies to focus at one or more



Table 5.16: Hypotheses Switching Behavior – By Model Component for all Subjects (11)

<i>Number of Hypotheses All Subject</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
242	Top-Down	N/A	49 18%	39 15%
144	Situation	56 21%	N/A	41 15%
175	Program	36 14%	45 17%	N/A

levels may be affected by task, expertise, amount of prior experience with the code, and strategy. We can say that hypotheses are related to model switching and when looked at collectively, switching occurs at a fairly even rate across models.

### 5.3.2.2 Hypotheses Switching – By Task

One of the questions we are interested in answering is whether the type of task affects hypothesis generated model switching behavior. Table 5.17 contains the number of switches caused by an hypothesis for each group of subjects working on a particular task. Appendix B, section B.2.2.1, provides more details on the results.

One important result from the analysis is that for two of the tasks (corrective maintenance, and general understanding) we can clearly see a strategy of employing hypotheses to generate a program and top-down model using the situation model as a bridge. This is important because this confirms Brooks' [9] theory of program comprehension. It states that programmers construct a mental representation by recreating mappings from the problem domain through several intermediate domains into the programming domain. This mapping process is driven by hypothesis generation. Our results confirm this general process for corrective maintenance and general understanding since these subjects often went from the top-down model (problem domain) to the program (programming domain) using the situation model as the intermediate domain.

Another question concerns whether task affects how often an hypothesis generated a switch. 40% of the adaptive maintenance hypotheses caused switches; corrective maintenance had 52% hypothesis generated switches; 39% of the enhancement hypotheses caused switches; and 49% of the general understanding hypotheses caused switches. These num-

Table 5.17: Hypotheses Switching Behavior – By Task

<i>Ttl. No. of Hypotheses: Adaptation</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
65	Top-Down	N/A	12 33%	3 8%
20	Situation	13 37%	N/A	2 6%
6	Program	3 8%	3 8%	N/A
<i>Ttl. No. of Hypotheses: Corrective</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
77	Top-Down	N/A	19 18%	16 15%
64	Situation	23 21%	N/A	17 16%
65	Program	12 11%	21 19%	N/A
<i>Ttl. No. of Hypotheses: Enhancement</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
18	Top-Down	N/A	0 0%	8 32%
5	Situation	1 4%	N/A	4 16%
41	Program	8 32%	4 16%	N/A
<i>Ttl. No. of Hypotheses: Understand</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
82	Top-Down	N/A	18 18%	12 12%
55	Situation	19 19%	N/A	18 18%
63	Program	13 13%	18 18%	N/A

bers are too close to conclude that task affects the frequency of hypothesis generated switches.

### 5.3.2.3 Hypotheses Switching – By Expertise

The next two questions are concerned with whether expertise affects how often hypotheses generate switches and whether hypothesis generated switching is influenced by expertise. Table 5.18 contains the number of switches caused by an hypothesis for each level of expertise. Appendix B, section B.2.2.2, provides more details on the results.

For subjects with language only expertise, 68% of all hypotheses involved a model switch. This is high compared to “domain only” and “language and domain” experts whose

Table 5.18: Hypotheses Switching Behavior – Expertise

<i>Ttl. No. of Hypotheses: Language</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
24	Top-Down	N/A	9 16%	7 12%
35	Situation	12 21%	N/A	11 19%
40	Program	5 9%	23 13%	N/A
<i>Ttl. No. of Hypotheses: Domain</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
110	Top-Down	N/A	16 17%	16 17%
53	Situation	19 20%	N/A	14 15%
44	Program	13 14%	17 17%	N/A
<i>Ttl. No. of Hypotheses: Language/Domain</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
108	Top-Down	N/A	24 21%	16 14%
56	Situation	25 22%	N/A	16 14%
91	Program	18 16%	15 13%	N/A

hypotheses generated switches at a much lower rate (46% and 45%, respectively). It might be that as expertise increases, strategies other than forming and resolving hypotheses may be used. For example, use of domain knowledge to make connections to the program and situation models is one such strategy. If the domain is familiar, the number of times a switch occurs (as a result of an hypothesis) goes down because they “just know” the answer without having to confirm it by looking at specific code. Thus, while experts make just as many hypotheses as non-experts, experts can make hypotheses in one model and not necessarily have to switch models to confirm. (Compare Figure C.38 of **AD2**, a domain only expert to Figure C.40 of **L1**, a language only expert. These figures of their hypotheses sequences show that **AD2** made approximately 16 hypotheses in the top-down model before switching to the program model, while **L1** had switched 8 times over the course of 16 hypotheses.) Domain novices may have to do much more switching as a result of an hypothesis. They do not have the experience necessary to be confident about answers without actually finding all evidence to resolve them.

Language only experts tend to switch more often from the program and top-down models into the situation model and from the situation model into the program and top-down models. Hypothesis sequence figures in Appendix C show this behavior was common for language only experts.

Domain only experts switched pretty evenly across all models. This may be because they have enough domain knowledge to make hypotheses at the domain level which they want to verify at the program model level. In order to do this, they can approach hypothesis resolution by decomposing the domain into lower-level expected functionality and using the situation model as an intermediate step into the program model.

Domain and language experts tend to switch more often between top-down and situation models or top-down and program models than between program and situation models. Again, this probably has to do with the ability to make direct connections between the top-down and program models.

#### **5.3.2.4 Hypotheses Switching – By Accumulated Knowledge**

The last question for hypothesis switching behavior is whether hypothesis caused switching behavior is influenced by amount of accumulated knowledge. Table 5.19 contains the number of switches caused by an hypothesis for each level of expertise. Appendix B, section B.2.2.3, provides more details on the results.

Subjects with *little* accumulated knowledge about the system they were working on switched models after generating an hypothesis more than those with *significant* accumulated knowledge. 48% of all hypotheses generated by subjects with *little* experience were followed by a switch to a different model. For those with *some* experience, 51% of hypotheses resulted in a switch. In contrast, those with *significant* knowledge had only 40% of all their hypotheses involved in a switch.

It is probable that those with *some* knowledge use hypotheses more efficiently than those with *little* knowledge because they already have partial models from previous experience. While our sample for subjects with *significant* knowledge did not give us sufficient information to draw conclusions, we would expect that they would have fewer overall hypotheses generated switches. We also expect they would be able to switch directly to and from the program model to the top-down model. This is, we believe, because by the time

Table 5.19: Hypotheses Switching Behavior – Accumulated Knowledge

<i>Ttl. No. of Hypotheses: Little</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
114	Top-Down	N/A	21 20%	18 17%
56	Situation	24 23%	N/A	12 12%
48	Program	15 14%	15 14%	N/A
<i>Ttl. No. of Hypotheses: Some</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
61	Top-Down	N/A	16 14%	14 13%
64	Situation	19 17%	N/A	23 21%
91	Program	13 12%	25 23%	N/A
<i>Ttl. No. of Hypotheses: Significant</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
67	Top-Down	N/A	12 23%	7 14%
24	Situation	13 25%	N/A	6 12%
36	Program	8 16%	5 10%	N/A

an engineer accumulates significant knowledge of the program, his mental model will be close to complete, therefore eliminating the need to generate many hypotheses that need to be resolved at a different level of abstraction. Also, familiarity with the code will allow direct jumping from top-down to program models without needing the situation model as a bridge.

### 5.3.3 HK-Ratio – Using Hypotheses and Knowledge to Indicate Expertise

As we have seen with action-types, expertise can affect understanding. However, in this study it was difficult to separate out the influence of prior experience with the code and prior experience in the domain. Thus, we were interested in discovering a method for gauging expertise based on the actions we saw. In particular, we were interested in the relationship between use of prior knowledge and making hypotheses.

This relationship can be expressed as the number of hypotheses per number of knowledge references:  $HK = \frac{\text{number of hypotheses}}{\text{number of knowledge references}}$ . If indeed the assumption holds that more experienced programmers make fewer hypotheses and instead use prior knowledge

during understanding tasks, taking the number of hypotheses and dividing it by the number of references to knowledge could serve as an indicator of expertise.

A low HK-ratio means fewer hypotheses than use of knowledge. Table 5.20 contains HK-ratios for each subject. It shows the total number of hypotheses and references to knowledge for each model. The HK-ratio is calculated and then a "Rank" is assigned by HK-ratio value. The lowest HK-ratio receives rank 1, the highest, rank 11. The table shows HK-ratios for each model level, situation/program model level together, and all model levels combined. This makes it possible to evaluate whether HK-ratios change by individual model level as well as across all model levels.

Table 5.20: HK-Ratio by Subject

<i>Subject Code</i>		<i>Program Model</i>	<i>Situation Model</i>	<i>Top-Down Model</i>	<i>Program &amp; Situation Model</i>	<i>Overall</i>
<b>AD1</b>	Hypo	3	14	23	17	40
	Know	19	18	51	37	88
	HK	.16	.78	.45	.46	.45
<b>AD2</b>	Rank	1	7	2	2	2
	Hypo	3	6	42	9	51
	Know	8	19	52	27	79
<b>C1</b>	HK	.37	.32	.81	.33	.65
	Rank	2	3	7	1	4
	Hypo	24	17	9	41	50
<b>C2</b>	Know	24	29	6	53	59
	HK	1.00	.59	1.50	.77	.85
	Rank	6	5	9	6	7
<b>C3</b>	Hypo	12	18	18	30	48
	Know	5	17	24	22	46
	HK	2.40	1.06	.75	1.36	1.04
<b>C4</b>	Rank	11	10	6	11	9
	Hypo	18	7	26	25	51
	Know	21	18	38	39	77
<b>C5</b>	HK	.86	.39	.68	.64	.66
	Rank	4	4	5	4	5
	Hypo	11	22	24	33	57
<b>C6</b>	Know	11	28	39	39	78
	HK	1.00	.79	.62	.85	.73
	Rank	6	8	4	7	6
<b>EN1</b>	Hypo	30	4	2	34	36
	Know	32	35	19	67	86
	HK	.94	.11	.11	.51	.42
<b>EN2</b>	Rank	5	1	1	3	1
	Hypo	11	1	16	12	28
	Know	10	7	31	17	48
<b>G1</b>	HK	1.10	.14	.52	.71	.58
	Rank	8	2	3	5	3
	Hypo	7	30	54	37	91
<b>G2</b>	Know	14	15	28	29	57
	HK	.50	2.00	1.93	1.28	1.60
	Rank	3	11	11	9	11
<b>L1</b>	Hypo	40	7	13	47	60
	Know	28	7	8	35	43
	HK	1.43	1.00	1.62	1.34	1.40
<b>L2</b>	Rank	10	9	10	10	10
	Hypo	16	18	15	34	49
	Know	13	26	15	39	54
<b>L3</b>	HK	1.23	.69	1.00	.87	.91
	Rank	9	6	8	8	8

Analysis of *Overall* HK-ratio in Table 5.20 produced interesting results. Three distinct groups are apparent:

- **EN1** (HK-ratio = .41), **AD1** (HK-ratio = .43), **EN2** (HK-ratio = .52), and **AD2** (HK-ratio = .63). All subjects were either domain or domain and language experts. Three of the four subjects (**AD1**, **AD2**, and **EN1**) had significant accumulated knowledge about the code they were working on.

- **C3** (HK-ratio = .65), **C4** (HK-ratio = .73), **C1** (HK-ratio = .85), **L1** (HK-ratio = .91), **C2** (HK-ratio = 1.04)

This group contains all the corrective maintenance subjects and one General Understanding subject. Also, both language only experts (**C1** and **L1**) were in this group.

- **G1** (HK-ratio = 1.40) and **G2** (HK-ratio = 1.74)

Both subjects worked on General Understanding tasks and both had domain and language expertise. Additionally, one subject had *little* prior experience with the code while the other had *some* experience.

Since the amount of accumulated knowledge seems to play an important part in code understanding, we wanted to determine if any patterns exist related to both the amount of accumulated knowledge and type of expertise. Tables B.23, B.24, B.25, and B.26 in Appendix B contain matrices for program, situation, top-down, and combined program and situation HK-Ratios grouped by accumulated knowledge and expertise. The overall HK-ratios (Table 5.21) for accumulated knowledge and expertise table are presented below. Additional analysis regarding the HK-Ratio can also be found in the appendix.

Table 5.21: Overall HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		5:C3 9:C2	11:G1 3:EN2
<i>Some</i>	7:C1 8:L1	6:C4	10:G2
<i>Significant</i>		4:AD2	2:AD1 1:EN1



Those with the most accumulated knowledge and highest level of expertise (EN1,AD1,AD2) have the lowest HK-ratio. They generate fewer hypotheses and make greater use of the knowledge they already have. Those with little or some accumulated knowledge and language only expertise (C1,L1) have higher rankings. They make minor use of prior knowledge because they do not possess enough of this prior knowledge.

While we are able to see some patterns based on accumulated knowledge and expertise, we also found some patterns based on task. The highest HK-ratios were seen with the two General Understanding subjects and is probably more influenced by task rather than expertise. It is interesting to note, however, that both subjects had either little or some accumulated knowledge, which could also be raising the HK-ratio. Making hypotheses while understanding new code may be a key strategy when the amount of accumulated knowledge is slight.

The corrective maintenance subjects were clustered around the middle ranks (between 5-9). Three of the four subjects were domain experts, but all had either *little* or *some* amount of accumulated knowledge. We conjecture that corrective maintenance requires a high number of hypotheses as a strategy for tracking down and fixing defects.

We also wanted to see how the HK-ratio corresponds to our original classification of expertise. Overall, subjects with low HK-ratios were originally classified as either domain only or domain and language experts and the majority of these had significant prior experience with the code. Subjects with little or some accumulated knowledge had much higher HK-ratios, except for one subject (EN2) who spent the majority of his time debugging a recent enhancement.

#### 5.4 The Role of Model Components in the Integrated Model – Level 1

One of the core tenets of the Integrated Model is that programmers build their mental model of the application at all three levels simultaneously. If this is true, we should find switches to and from all models. (Top-down understanding would preclude switches from program and situation models up to the domain model, while bottom-up understanding would not show switches from the domain model down to the situation and program models.)

Although the integrated model predicts switching between all levels of abstraction, we would also expect to see preferences for specific models based on the size of the software to be understood. (For example, the nature of the program model is to look at code line by line, which is not feasible for software hundreds of thousands of lines long.)

Table 5.22 (itself a refinement of Table B.28 in Appendix B) shows that we do indeed find multi-directional switching to and from all models. As before, there are other obvious questions related to the role of switching and code size:

- Does model switching show specific patterns depending on expertise, accumulated knowledge, or type of task?
- Does the amount of time spent in each model component depend on the size of software with which the programmer is working?

#### **5.4.1 Switches Between Models**

Switches are defined as a change in focus from one level of abstraction to another. A switch occurs from any one of the three models into any other model except the source. (It makes no sense to switch into the model if you are already there.) Previously we presented results on switches caused by hypotheses. Hypotheses are a specific type of action. In this section we report on switches caused by any action. We do not distinguish between types of actions. A switch occurs when the subject is performing an action in one model and the subsequent action occurs in a different model. Table 5.22 shows model switches by task and expertise.

Table 5.22: Action Switches – Absolute & Percent of Total Switches by Task & Expertise

Task	Expertise	Model	Model Switches – All Subjects		
			Top-Down Model	Situation Model	Program Model
Adaptive (1 subject)	Domain Only (Total switches = 89)	Top-Down	N/A	14 16%	15 17%
		Situation	15 17%	N/A	15 17%
		Program	14 16%	16 17%	N/A
Adaptive (1 subject)	Domain & Language (Total switches = 132)	Top-Down	N/A	28 21%	33 25%
		Situation	28 21%	N/A	5 4%
		Program	34 26%	4 3%	N/A
Corrective (1 subject)	Language Only (Total switches = 118)	Top-Down	N/A	14 12%	14 12%
		Situation	18 15%	N/A	29 25%
		Program	11 9%	32 27%	N/A
Corrective (3 subjects)	Domain Only (Total switches = 444)	Top-Down	N/A	53 12%	79 18%
		Situation	68 15%	N/A	83 19%
		Program	64 14%	97 22%	N/A
Enhancement (2 subjects)	Domain & Language (Total switches = 299)	Top-Down	N/A	15 5%	77 26%
		Situation	28 10%	N/A	51 17%
		Program	64 21%	64 21%	N/A
General Understanding (1 subject)	Language Only (Total switches = 134)	Top-Down	N/A	9 6%	17 13%
		Situation	13 10%	N/A	39 29%
		Program	13 10%	43 32%	N/A
General Understanding (2 subjects)	Domain & Language (Total switches = 286)	Top-Down	N/A	21 7%	62 22%
		Situation	34 12%	N/A	53 19%
		Program	49 17%	67 23%	N/A

### 5.4.1.1 Switches Between Models – Totals By Tasks

The first question we are interested in answering is how task influences switching behavior. Table 5.23 shows model switching data based on task.

Table 5.23: Action Switches – Absolute & Percent of Total Switches by Task

Task	Model	Model Switches – All Subjects		
		Top-Down Model	Situation Model	Program Model
Adaptive (2 subjects) (Total switches = 221)	Top-Down	N/A	42 19%	48 22%
	Situation	43 19%	N/A	20 9%
	Program	48 22%	20 9%	N/A
Corrective (4 subjects) (Total switches = 562)	Top-Down	N/A	67 12%	93 17%
	Situation	86 15%	N/A	112 20%
	Program	75 13%	129 23%	N/A
Enhancement (2 subjects) (Total switches = 299)	Top-Down	N/A	15 5%	77 26%
	Situation	28 10%	N/A	51 17%
	Program	64 21%	64 21%	N/A
General Understanding (3 subjects) (Total switches = 420)	Top-Down	N/A	30 7%	79 19%
	Situation	47 11%	N/A	92 22%
	Program	62 15%	110 26%	N/A

Switching between models to build a multi-tiered mental model of a program is important for all the tasks we studied. Switching during adaptation occurred predominately between program and top-down models and top-down and situation models (in both directions), rather than between program and situation models. We can hypothesize that for adaptive tasks like porting, high level information tied directly to code is more important than having a thorough low-level understanding as embodied in a complete program and situation model.

Switches during corrective maintenance occur slightly more often between program and situation models. For corrective maintenance, having that low level information is important. It helps to more effectively track down defects and understand them. Enhancement trends were difficult to define, however, they had much in common with the

corrective tasks. Finally, General Understanding tasks are most likely to be affected by overall strategy. A line-by-line understanding strategy results in a lot of switching between program and situation models only. A more opportunistic strategy can result in a lot of program to top-down or program to situation model switches. What our subjects had in common was a significant lack of top-down to situation model switches.

One other conjecture can be postulated. Programmers, regardless of the task, switch between all three levels. In only four cases did the subjects switch between a particular pair of models under 10% of the (switching) time. Two of these cases involved switches between program and situation models and the other two were switches from the top-down model into the situation model. This leaves 20 cases where they switched at a rate of 10% or higher between pairs of models. (See Table B.28.) Of course, expertise must play a part and we will examine this influence in the next section.

#### 5.4.1.2 Switches Between Models – Totals By Expertise

The next question we are interested in answering is whether expertise affects switching behavior. Table 5.24 presents model switching data by level of expertise.

Table 5.24: Action Switches – Absolute & Percent of Total Switches by Expertise

<i>Expertise</i>	<i>Model</i>	<i>Model Switches – All Subjects</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
Language (2 subjects) (Total switches = 252)	Top-Down	N/A	23 9%	31 12%
	Situation	31 12%	N/A	68 27%
	Program	24 10%	75 30%	N/A
Domain (4 subjects) (Total switches = 533)	Top-Down	N/A	67 13%	94 18%
	Situation	83 15%	N/A	98 18%
	Program	78 15%	113 21%	N/A
Language & Domain (5 subjects) (Total switches = 717)	Top-Down	N/A	64 9%	172 24%
	Situation	90 13%	N/A	109 15%
	Program	147 20%	135 19%	N/A

Several hypotheses can be postulated from the analysis of switching behavior based on expertise. For engineers with language only expertise, most switches were between

program and situation models (in both directions.) Lack of domain knowledge means programmers do not have the tools necessary to easily make connections from the domain to the code or vice versa. They predominantly build their mental models at the program and situation model levels until they acquire enough expertise in the domain to be able to construct a complete top-down model. Domain experts switch between all three models in all directions. One explanation for this is that domain experts have the knowledge necessary to build a high-level representation but need the situation model as an intermediate domain in which to decompose known components in the top-down model into smaller components that are closer to the code. Finally, domain and language experts switch more often between the program and top-down models in both directions. Domain and language experts have both the domain and language knowledge that allows them to make direct connections between program and top-down models without the help of an intermediate situation model.

#### 5.4.1.3 Switches Between Models – Totals By Accumulated Knowledge

Accumulated knowledge might make a difference in how programmers switch between models (i.e. make connections between models) during a maintenance task. For example, if the design of a system is completely understood, the programmer may be switching predominantly between the top-down model and the program model. To investigate how accumulated knowledge affects switching behavior, we grouped the switching data by levels of accumulated knowledge (*Little*, *Some*, and *Significant*).

Table 5.25 contains model switching data for accumulated knowledge.

The average number of switches per programmer is:

- *Little* = 175.25
- *Some* = 122.0
- *Significant* = 104.3

Thus, there appears to be a declining trend as accumulated knowledge increases and this may be due to a lesser need for establishing cross references between models. Looking in more detail, at the models between which programmers switch, switching between program and situation models (in both directions) occur at different rates depending on

Table 5.25: Action Switches – Absolute & Percent of Total Switches by Accumulated-Knowledge

Accumulated Knowledge	Model	Model Switches – All Subjects		
		Top-Down Model	Situation Model	Program Model
Little (4 subjects) (Total switches = 701)	Top-Down	N/A	51 7%	161 23%
	Situation	83 12%	N/A	123 18%
	Program	128 18%	155 22%	N/A
Some (4 subjects) (Total switches = 488)	Top-Down	N/A	54 11%	69 14%
	Situation	67 14%	N/A	114 23%
	Program	58 12%	126 26%	N/A
Significant (3 subjects) (Total switches = 313)	Top-Down	N/A	49 16%	67 21%
	Situation	54 17%	N/A	38 12%
	Program	63 20%	42 14%	N/A

the amount of accumulated knowledge. For those with *little* accumulated knowledge these switches occur at a rate of 40%; those with *some* accumulated knowledge switch between program and situation model 49% of the time; and those with *significant* accumulated knowledge switch only 26% between program and situation models.

Switches between program and situation models indicate cross references between these two models and building of an implementation understanding of the code. In this regard, subjects with *little* accumulated knowledge and those with *some* accumulated knowledge behave similarly, although those with *some* accumulated knowledge show a more even distribution among all possible switches. As knowledge increases, the implementation may be more fully understood, explaining the drop in switches between program and situation models.

Subjects with *significant* accumulated knowledge switched most often either between program and top-down models or between top-down and situation models. Someone who has *significant* experience with a particular piece of code will have reasonably complete mental representations at all levels. Therefore, switching will be goal-based and focused on areas of change. This is why, for example, when adding new functionality, connections

between the new functionality at the top-down level and actual code location at the program model level will be predominant.

#### 5.4.2 The Role of Code Size on Model Building

We are now at a point where we can investigate quantitatively whether the size of the code to be understood makes a difference in how programmers approach their task and what this difference might be. The first analysis would identify the number of actions at each model (comprehension level) organized by the code size as in Table 5.26.

The table is organized according to code size, ranging from *One module* to an entire *System*. Within each size level, columns are arranged by expertise and report on specific subjects. The first sub-column contains the subject identifier followed by the component model code ( C1-Prg means subject C1, program model; Sit for situation model; TD for top-down model). The second sub-column contains the number of references to the component model for that subject. At the far left of the table, the first two columns (*Model* and *% of Ttl. Num*) show the total number of references to the model component and frequencies over all levels of expertise for a given code size. For example, the total number of program model references for a code size of *One Module* is 482 which is the sum of C1's, L1's, and G2's program model references. Below the total reference count we also give a relative frequency (percentage).

Table 5.26 contains some very interesting results. If the size of the component is small enough to understand at a low level of detail, it makes sense to spend most of the time in the program model. On the other hand, if the code size is large (AD1 worked with a system of about 90,000 lines of code) then understanding occurs preferably at a higher level of abstraction. At the highest level is domain knowledge and ideally programmers should proceed with understanding at this level if the code is large. Obviously then, sufficient knowledge at the domain level is very important for large systems, since programmers refer to this level more during understanding.

#### 5.5 Information Needs

So far we have identified types and frequencies of actions as elementary components of what programmers do. Many of these actions reflect a search for information to include in the mental model of the application. We now analyze what these information needs



Table 5.26: Frequencies of References to Model Components (All Subjects)

		<i>Expertise ⇒ Component Size to Understand ↓</i>	<i>Language Expert</i>		<i>Domain Expert</i>		<i>Language &amp; Domain Expert</i>	
<i>Model</i>	<i>% of Ttl. Num</i>							
Program	530	One Module	C1-Prg	143			G2-Prg	243
	55%		L1-Prg	144			G2-Sit	33
Situation	283		C1-Sit	160			G2-TD	59
	29%		L1-Sit	90				
Top-Down	155		C1-TD	46				
	16%		L1-TD	50				
Program	576	Several Modules			C3-Prg	160	EN1-Prg	193
	56%						EN2-Prg	223
Situation	153				C3-Sit	42	EN1-Sit	58
	15%						EN2-SIT	53
Top-Down	305				C3-TD	119	EN1-TD	32
	29%						EN2-TD	154
Program	322	Whole Program			AD2-Prg	61	G1-Prg	112
	29%				C2-Prg	149		
Situation	216				AD2-Sit	47	G1-Sit	78
	20%				C2-Sit	91		
Top-Down	561				AD2-TD	220	G1-TD	222
	51%				C2-TD	119		
Program	153	System			C4-Prg	85	AD1-Prg	68
	22%							
Situation	144				C4-Sit	101	AD1-Sit	43
	21%							
Top-Down	397			C4-TD	171	AD1-TD	226	
	57%							

are. Our motivation is that if tools or documentation could provide this information, this could benefit the comprehension process.

Naturally, we would also like to know what the most frequently encountered information needs are. Meeting those provides the biggest benefit. It would also be useful to know whether specific types of tasks have preferred information needs. A third question relates to whether or not there are differences in information needs based on experience level or accumulated knowledge with a piece of software.

Information Needs are data necessary during a comprehension task. They can be determined directly from statements about needed information or indirectly through actions (such as looking for specific information) or when the programmer could obviously benefit if she had the data. Appendix B, section B.4 contains more details on the results.

### 5.5.1 Frequencies for All Subjects

To show overall trends for all subjects, Tables 5.27, 5.28, and 5.29 contain information needs (ordered by frequency). The information needs table contains four columns.

The first holds the code used in analyzing the protocols for information needs. The second describes the information need. The third lists the action codes (associated with a specific model) for which the information is needed. The last column lists how often the information need occurred in all 11 protocols.

Table 5.27: Information Needs Frequencies: All Subjects

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Info Need Frequencies</i>
I7	Domain concept descriptions	OP1	85
I61	Connected domain-program-situation model knowledge	SYS7	70
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	47
I9	List of browsed locations	OP2, OP4, SYS2 OP4	34
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	23
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	23
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	21
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	17
I3	Highlighted begin/ends of control blocks	SYS3, SYS7, SYS8, SYS10, SYS24	17
I22	History of past modifications	SYS3	17
I73	Bug behavior isolated	SIT1	17
I14	Call Graph Display	OP1, OP2, SYS3	16
I68	List of issues/decisions considered during design	SYS12	16
I24	List of executed statements and procedure calls, variable values	SYS8	14
I42	Utility functions definitions and comments explaining why it was rewritten	OP2	14
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	12
I17	Location of desired code segment	SYS1, SYS2	10
I6	Location and description of library routines and system calls	OP1, OP2, OP20, SYS11	9
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	9
I58	High-level description of how code is laid out	OP3	9
I67	Location of where to put changes	SYS11	9
I20	Documentation list and location	OP1	8
I44	List of routines that do most of the domain-type work	OP3	8
I72	Good direction to follow given what is already known, possible program segments to examine	SYS10	8
I16	Naming conventions separated by system or library objects that use them Rules used for new naming new procedures	OP3, SYS3, SYS8 SIT7	7
I36	Sequence of locations where ID is used	OP2	7
I53	IFDEF'd behavior, conditions under which the branch is taken or not	SIT2	7
I66	Expected program state, e.g. expected variable values when procedure is called	SYS11	7
I74	Good description of the bug and why and how other fixes were done	SIT1	7

Table 5.28: Information Needs Frequencies: All Subjects (continued)

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Info Need Frequencies</i>
179	Ripple Effect – Procedure affected by change, include port affects, dependencies in in Make file		7
111	Language conventions that are different from standard conventions Language enhancements	SYS1, SYS3, SYS10	6
112	Environment, global, local scope	SYS3	6
118	I/O parameters, definitions, examples of calls with explanation, what it is used for	SYS3, SYS5, SYS8	6
162	Predefined (constant) variables and values	SYS8	6
170	State of system when crashed	SYS18	6
113	Conditions under which a branch is taken or not. Include variable values	SYS3	5
129	Naming conventions for program	SYS1, SYS3	5
130	Where variable is toggled, when and why, where passed to and why	SYS2, SYS12	5
138	Nesting level of a particular procedure	SYS2, SYS7	5
139	Main program location	OP1	5
141	Call graph with extraneous information not relevant elided	OP2	5
148	Code formatted in expected way	SYS3	5
178	Dynamic function return values		5
119	Acronym definitions	SIT1, SIT7	4
125	Exact location to set breakpoint	SYS18	4
137	Language definitions,e.g reserved words, instruction defs, for C, Pascal, etc.	SYS8, SYS18	4
145	List of internal support routines	OP3	4
147	List of identifiers and domain concepts that are important	OP4	4
149	File name of current file	SYS3	4
155	Domain Simulation	SIT2	4
171	Concise error description	SYS20	4
176	Available tools to aid in understanding		4
110	Function call count	OP1, SYS2	3
121	Organized functions into categories in which functions are related	OP2	3
126	All include file definitions and uses	OP1, SYS5	3
128	List of all routines with initialization code	OP2, SYS2	3
152	User defined code block identification	SYS7	3
163	All definitions and uses of a variable prioritized...so more important uses show up first	SYS10	3
164	System configuration for rebooting	OP6	3
165	Assembly Language Code Segment number (Machine Code)	SYS2	3
18	List of system calls based on specific naming convention	OP1	2
140	List of capabilities that may need to be provided	OP1	2
146	What kind of architecture dependent code currently exists and is functional what architecture hooks exist so code can be easily added.	OP3	2
151	What happens to read in data	SYS5	2
154	List of experts	SYS10	2
159	How a variable is passed into a procedure, e.g. by value, address, etc.	SYS1	2

Table 5.29: Information Needs Frequencies: All Subjects (continued)

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Info Need Frequencies</i>
I77	Data-flow trace		2
I35	Any difference between variable name and use	SYS8	1
I56	The include file a particular structure is defined in	OP2	1
I69	Possible errors and severities, case that always need safety checks	SYS17	1
I80	Program execution sequence, how to use (e.g. man page)		1

Overall, *I7* (Domain concept descriptions) and *I61* (Connected domain-program-situation model knowledge) were the most frequent information needs. Domain concept descriptions include high-level information, for example, operating system concepts. Connected model knowledge is information that allows cross-referencing from one model to another. For example, if a chunk of code is labeled “sort”, the label can be viewed as the connection between program and situation models. It is not surprising that these two types of information are needed significantly more often than other types of information. This strengthens the hypothesis that programmers try to build different levels of abstract views of the program. The reason we saw the need for domain concepts and connected model information so frequently could be because it is the type of information that is not easy to get from existing tools and technology. The usual scenario is that any information above the program model level has to be searched for and connections made manually.

Another important information need, *I43* (General classification of routines) is also related to understanding domain concepts and cross-referencing between models. Understanding that a group of procedures are related helps not only to build connections between models, but also to direct focus during the comprehension task. When a programmer realizes that a group of functions are related, she can either ignore it if the functions are not relevant to the task or, she can focus on them if they are important.

Recalling recently browsed information (*I9*) is also important. The ability to scroll back and forth between data reduces cognitive overload. A common example occurs when a programmer skims data she does not fully understand. She needs to revisit the information when other data triggers a recall and subsequent comprehension of the previously skimmed data.

The majority of the remaining significant information needs are also not surprising. Things like definition and uses of identifiers (*I4* and *I1*), data structure format (*I5*) and function calling structure (*I2* and *I14*) are all related to major characteristics of most programs. Access to this type of information is essential to understanding how the program works.

### 5.5.2 Information Needs – By Task

Next, we consider whether there are any information needs that predominate for a specific task. Appendix B, section B.4.1 contains Tables B.31 and B.32 which report on information needs grouped by task type.

The most frequent types of information needed by the adaptive maintenance subjects were *I7* (Domain concepts), *I22* (History of past modifications), and *I79* (impact of changes). Nothing can be concluded about information needs for the adaptive maintenance subjects, except that these two subjects had the fewest number of information needs than any of the other tasks. This is probably a function of both expertise and accumulated knowledge, since both were domain experts and both had significant knowledge about their software.

For corrective maintenance, the three most frequent information needs are *I7* (Domain Concepts), *I61* (model connected information), and *I4* (location and uses of identifiers). Other important information needs are *I9* (List of browsed locations), *I2* (Function called by), *I4* (call graph display) and *I73* (bug behavior isolated). Understanding such code level information is crucial to understanding a defect.

The enhancement subjects' most common information needs are *I9* (List of browsed locations), *I4* (Location and uses of identifiers), and *I67* (code location to put changes). Both subjects needed to get back to a place in the code where they had recently been and wanted to know the location more often than any other type of information. Similar to the adaptive maintenance subjects, it is difficult to draw further conclusions for enhancement since there were a low number of total information needs.

General Understanding requires domain knowledge (*I7*) and connected model information (*I61*) more frequently than other information. When there is no specific goal other

than to understand the program (for future maintenance), both types of information helps to build mental representations at the domain level as well as at other levels.

Tasks like corrective maintenance and general understanding seem to have more need of information than tasks like adaptive maintenance or enhancement. While our sample size was small, each group had a similar number of information needs with exceptions due to programming session time. The average number of references per subject for corrective maintenance was 73.5 and for general understanding is was 81.67. The average number of references for adaptive maintenance was 34 and for enhancement it was 48. We could hypothesize that this difference lies in the requirements of the tasks. Debugging and general understanding are investigative tasks and need information to solve problems and misunderstandings. Adaptive maintenance and enhancements tasks are less investigative and more action-oriented. When performing some action, the information is presumably already available rather than sought after.

### 5.5.3 Information Needs – By Expertise

The next question to consider is whether expertise makes a difference in the types of information needs most frequently encountered. Tables B.33 and B.34, in section B.4.2, report on information needs grouped by expertise.

The top two information needs are the same regardless of expertise: *I7* (domain concepts) and *I61* (connected model information). Language only experts also need code level information (*I2* is the third most frequent information need). On the other hand, Domain only and Domain and Language experts' third most frequent information needs are *I4* (Location and uses of identifiers) tied with *I9* (list of browsed locations) for Domain and Language experts. Three of four subjects with only Domain expertise were working on debugging tasks, which could explain the high need for information about identifiers at the code level. For Domain and Language experts, two subjects (**G2** and **EN1**) had 70% of the references to the *I4* needs. Both subjects spent the majority of their time at the program model level. At all levels of expertise, information is needed for both the domain and program models. In essence, there is very little difference in information needs between levels of expertise.

#### 5.5.4 Information Needs – By Accumulated Knowledge

Finally, we would like to investigate whether information needs change significantly depending on amount of accumulated knowledge. Tables B.35 and B.36 (section B.4.3) report on information needs clustered by amount of accumulated knowledge. First, do subjects with less accumulated knowledge have as many information needs as those with more accumulated knowledge? Engineers with *little* accumulated knowledge need more information than those with *some* or *significant* experience with the code. Each subject with *little* accumulated knowledge had an average of 80 information needs. Those with *some* prior experience had an average of 67.5, and those with *significant* accumulated knowledge had an average of 37.67. The types of information range from one time understanding needs (i.e. call graphs and general classifications of functions) to on-going development of models and the connections between them. People with *some* accumulated knowledge may have similar needs to those with *little* experience, depending on how much more experience they have. In this regard, our classification of subjects into the *some* category may have been too broad. Also, because three of the four subjects in this group worked mainly at the program model level, information needs tended to be at that level, i.e. they had more total program model related information needs (such as *I1*, *I2*, *I3*, *I4*, *I5*, *I24*, and *I66*) than total domain level information needs. However, they did have a lower average number of information needs per subject than the *little* accumulated knowledge group and a higher average number than those with *significant* experience. Finally, those with *significant* accumulated knowledge already have a near complete mental representation at all levels with the appropriate connections between. Their information needs are more likely to be related mostly to the task they are working on and less on building connections between models. They are also likely to have less information needs in general.

#### 5.6 Tool Capabilities

Our motivation in developing Tool Capabilities is to help maintainers by providing useful tools that meet their needs. Tool Capabilities describe functionality that needs to be provided by software development and maintenance tools if these tools are to provide support for building top-down, situation, and program model mental representations of programs. Tool capabilities were derived from the most important information needs found



in the protocols of 11 subjects. Each information need was analyzed and capabilities were derived for meeting that need. In most cases, a single capability description addresses multiple information needs. Table 5.30 summarizes the tool capabilities that would meet programmers' information needs. It contains four columns. The first column contains a code for designating a specific tool capability. The second describes the tool capability. The third contains the information needs that were used to derive the particular capability. The last column contains the total number of instances where this tool would have helped during all programming sessions. This was calculated by summing the total number of associated information needs in column three for all subjects.

*T1* is the most important tool capability. Definition of program and system components are very common types of information needed during program understanding. Components can be defined, both within a program or outside the program in the surrounding environment. While there are plenty of software development tools that can provide the basic definition of an object<sup>1</sup>, building mental representations at different levels of abstraction is usually not supported.

The second most important capability, *T2*, is related to *T1*. Not only are definitions of objects important, but also their locations and uses. Most software development tools include this capability together with a method for getting to the specific instance of an object. Scoping and filtering capabilities, however, can vary. By scoping we mean the ability to indicate where to look for objects, i.e. within a particular file, function, directory, set of directories, libraries, etc. Filtering takes a set of results after scoping is applied and allows the user to utilize a method for culling them down to a usable list. This is especially important in cases where there are hundreds of references to a single object.

The next important tool capability is *T9*, "On-line text books" and "documentation for domain and situation model concepts". This is because knowledge about the domain was the most important information need we saw for the 11 subjects. This is also probably one of the most difficult functionality to build into a tool because it necessarily has to be domain specific. One way would be to provide links into textbooks based on keywords. The

---

<sup>1</sup>By object, we mean any artifact in a program, software system, or environment. This is not necessarily an object-oriented definition of an "object".

availability of project documents on-line (*T7*) is also an essential tool capability. Ideally, these documents would be linked together based on requirements and also perhaps domain model components. The ability to search on domain-based keywords would help to build the mental representation at different levels of abstraction.

Table 5.30: Tool Capabilities Table

<i>Code</i>	<i>Tool Capabilities</i>	<i>Information Need</i>	<i>Frequency All Tasks</i>
T1	Pop-up object definitions with hypertext capabilities following domain, situation, and program model links. Objects are program elements (classes, variables, instances, macros, constants, etc.), files, build support, directories, user defined groups of objects, resources, and environment variables	I1, I5, I7, I12, I43, I45, I61, I62	234
T2	Cross reference objects with keyword search capabilities for list of all definitions/uses. Click on to auto-jump to specific instance in code. Include scoping capabilities and regular expressions to filter.	I4, I6, I17, I26, I30, I39, I49, I63	86
T3	Call graph with user definable elision. Pruning and filtering based on user definable parameters. Ability to annotate with domain concept labels.	I14, I26, I38, I41, I58	38
T4	History of browsed locations, action sequences Multi-level undo. Backward/forward scrolling through actions, book-mark capabilities	I9, I72	42
T5	Grouping capabilities for objects. On-line code with grouping capabilities for finding and formatting lists of objects. Ability to assign group attribute to any set of objects, include group of groups. Ability to perform same queries available on objects.	I8, I16, I21, I28, I42, I44, I45	41
T6	Directory structure graph, project structure graph, file graph with query capabilities and regular expression filtering/scoping.	I26, I27, I49	19
T7	On-line project documents, including list of available documents. Keyword and regular expression search capabilities. Naming conventions tied to domain concepts. Hyper-text link from source code to project documents. Document includes requirements/specifications, design, test plans and cases, design decisions, defect reports, man pages, past modification history.	I16, I19, I20, I22, I29, I47, I67, I68	70
T8	On-line language manuals with query capability for language constructs. Emphasize conventions for uses. Provide extensive examples of use. Query capabilities.	I11, I32, I37	19
T9	On-line text books or documentation for domain and situation model concepts. Provide query capabilities	I7	85
T10	Data-flow and Def/Use graphs with query capabilities	I24, I30, I36, I53	33
T11	On-line error reporting with bug behavior isolated. Include any prior fixes, and system configuration to recreate the bug	I64, I70, I71, I73, I74	37
T12	Smart debugger with breakpoint assistant - Auto set according to code changes - Query capabilities on breakpoints - Trace capabilities by object/group name - Profile of executed statements/variable value	I24, I25, I36, I66, I78	37
T13	Impact analyzer, support for "what if" changes before actual changes are made.	I66, I79	14
T14	Editor functionality: - Click-on automatic Begin/End code blocks - Annotation/notepad capabilities for all docs - User definable code re-formatter	I3, I48, I52	25
T15	Chunker: ability to label chunks of code with meaningful identifier and elide from view, showing only the label.	I61	70
T16	Object calls/called by graphs	I2	21
T17	Branch flow analysis, IFDEF analysis	I13, I53	12

The fifth most frequent tool capability (*T15*) is the ability to chunk code, label each chunk with a meaningful identifier, and hide the details of the code from view, keeping only the label visible. This is especially important during program model building for abstraction into the situation model.

The ability to browse recently acquired information or to repeat actions (*T4*) is the sixth most important tool capability. A common situation for software engineers arises when they have performed a sequence of steps (perhaps following the data flow of a single variable) and want to get back to a previous state because something they saw suddenly makes sense. *T4* is also useful when a series of actions needs to be repeated several times, or a series of actions needs to be undone because the results were unexpected. Tools that provide multi-level undo for actions, browsing and book-mark capabilities to see where you have been, and an action capture (perhaps similar to shell-scripts, yet more user-friendly) are valuable. This is especially true in light of the popularity of browsers like Netscape.

Another very important tool capability, and one that is not commonly found in most software development tools, is the ability to group (chunk) program and system components. This is basic for developing higher levels of abstraction for program code. Functions, classes, and files are examples of program components that can have relationships among components of the same kind as well as between different types. For instance, a set of functions may be the implementation of specific functionality in the program. The capability to group (chunk) related components together supports development of higher level abstractions and can help in making understanding more efficient. This is the case when a component group is identified as not relevant to the task at hand. The engineer will know not to waste time in understanding that group of components. It is also important to be able to perform queries and searches based on these groups as well as provide the user with a method for defining component groups.

## 5.7 Core Competencies

Information Needs relate to expertise and knowledge in the sense that “if I know it, I don’t have to look for it”. How then would one provide the programmer (who is new to an application and possibly the application domain) with useful material that

would “jumpstart” her to develop the necessary core competency to maintain a particular software product successfully?

Our analysis found a variety of information needs and useful tool capabilities. We can abstract and describe the evidence as part of the minimum knowledge necessary to perform maintenance tasks. We can also partition the knowledge into groups of program, situation, and top-down related structures. Such knowledge could be defined as *core competencies*.

These core competencies are determined by looking at the most frequent information needs and answering a couple of questions. First, how useful, in general, would this information be to people new to the code? How useful to those more experienced with the code? Second, how difficult is it to extract the information and present it in a form that someone new to the code can quickly assimilate? Table 5.31 presents an attempt to describe core competencies associated with the most frequent information needs. It is organized by model level (top-down, situation, and program). Table 5.31 contains core competencies like *Product Specific Knowledge* and *Variable & Component Names*. Static analyzers in existence today can extract important program identifiers quite easily. *Man* pages are a simple way to extract usage information for command line functions and programs. Extracting appropriate groups of related program components (see Information Needs I16, I21, I28, I42, I44, and I45) is a harder task and must be done by experts who are very familiar with the code. Other information needs, such as I66 (Expected program state) are not worth the effort to include in a set of core competencies. Someone new to the code can simply watch the program run using a debugger. The *Comments* columns not only describe in more detail what the competency is but also how it can be acquired.

While our results are incomplete, we have an indication of the types of information that will be useful for maintenance engineers based on the results from eleven protocols.

Table 5.31: Core Competencies by Model Component

<i>Competency</i>	<i>Comments</i>
<i>Domain Related Competencies</i>	
<b>Product Specific Knowledge</b>	Commands and Use (e.g. HPUNIX vs.UNIX operating system commands) System Configuration (e.g. How to configure the system for test or bug reproduction.)
<b>Area Knowledge</b>	Standard, product independent information (e.g.Operating systems principles.) Prior Experience Formal instruction
<b>Architecture</b>	Structure (e.g. An operating system as components: Process, I/O, memory,and file management. Include directory structure) Interconnections (e.g. How are they related functionally.)
<b>Cross references</b>	Within the domain including where to find the information we need(e.g the <i>expert</i> or specific text book. Design issues & resolution )
<b>Links to other models</b>	connect functionality to algorithms and code
<b>Domain Groups</b>	what are the functionally related groups of program/system components?
<b>Key terms</b>	<b>VERY IMPORTANT</b> This guides understanding and ties the domain model to the situation and program models. (e.g. PROCESS MANAGER using ROUND ROBIN scheduling algorithm.)
<i>Situation Related Competencies</i>	
<b>Algorithms and data structures</b>	Language independent, detail design level (e.g. functional sequence of steps in the round robin algorithm or a graphical representation of a process queue.)
<b>Detailed design</b>	Close to code, but language independent. Specific product information in functional terms.
<b>Links to Program &amp; domain information</b>	Find design rational. Connect algorithm to purpose of application.
<b>Conventions</b>	Use the same terms across comprehension & models. <b>THESE ARE THE KEY TERMS!</b>
<b>Cross reference levels of information with connections to other models</b>	Also within the same situation model level
<b>Situation Model Groups</b>	What algorithms and objects are related? How do they map into Domain and program model groups?
<i>Program Model Related Competencies</i>	
<b>Variable &amp; component names</b>	KEY TERMS : meaningful mnemonic and acronyms for symbols.
<b>Links to situation and domain</b>	Capability to follow beacon to design and domain information.
<b>Critical sections of code identified</b>	Focus attention; improve efficiency
<b>Formalized beacons</b>	Focus attention; improve efficiency
<b>Cross references &amp; Connections</b>	Back to the situation and domain levels.
<b>Program Model Chunks</b>	These are chunks. How do these chunks map into the situation model groups?

## 5.8 Conclusions

Several hypotheses were stated throughout Chapter 5. We present the most important conclusions for actions, processes, hypotheses, switching, and information needs.

### 5.8.1 Actions

- Use of previously acquired knowledge and generating hypotheses are within the top four action types for all models, making them highly important actions for understanding at any level.
- Once a significant amount of domain knowledge is acquired, it can be used or not used depending on task and focus. For example, EN1 was both a language and domain expert with significant experience with the code. However, he did not have many references to the top-down model because his focus was on code he had recently written rather than building a domain representation. At the time of the programming session, he had already constructed the top-down model of the new code and did not need to refer to it very often.
- When domain knowledge does not exist, mental representations are concentrated toward program and situation models. When domain knowledge exists, mental representations can exist at all levels of abstraction.

### 5.8.2 Processes

- At least three comprehension processes exist for understanding code: systematic, opportunistic, and systematic/opportunistic.
- In systematic processes we see well-defined sequences of actions repeated during understanding. These sequences can be aggregated until an overall process of code comprehension for specific maintenance tasks are identified.
- Opportunistic processes consist of triad sequences of goals-hypotheses-actions. They are opportunistic in that hypotheses are resolved based on perceived relevance. Experts who use opportunistic processes have an arsenal of possible hypotheses and supporting actions they employ to understand code.

- Systematic/opportunistic processes include a combination of systematic and opportunistic understanding.

### 5.8.3 Hypotheses

- HK-Ratios
  - Programmers with the highest amount of accumulated knowledge and expertise have the lowest HK-ratio. They generate fewer hypotheses and make greater use of the knowledge they already have.
  - Programmers with *little* or *some* accumulated knowledge generate more hypothesis and make minor use of prior knowledge because they do not have the knowledge to exploit.
  - Corrective maintenance requires a high number of hypotheses as a strategy for tracking down and fixing bugs.
  - The relationship between hypothesis and use of prior knowledge is a good indication of expertise. Overall, subjects with low HK-ratios were originally classified as either domain only or domain and language experts and the majority of these had significant prior experience with the code. Subjects with little or some accumulated knowledge had much higher HK-ratios, except for one subject (EN2).
- Hypothesis Types
  - Corrective maintenance hypothesis generation may be predictable in that at all levels, hypotheses about correctness dominate the type we see.
  - The lack of expertise appears to affect the number of hypotheses generated within a particular model. Language only experts tend to generate mostly program model hypotheses, domain experts make the most top-down hypotheses, and domain and language experts generate a number of program and top-down model hypotheses that fall somewhere between the other two groups.
  - Amount of accumulated knowledge affects the types of hypotheses generated. Domain model connections become increasingly more important with less accumulated knowledge. Program model hypotheses drop off with increases in



accumulated knowledge. The majority of the situation model hypotheses are made by engineers with some accumulated knowledge. They tend to use the situation model as a bridge from the code to the domain.

- Hypothesis Generated Switching Behavior
  - Programmers use hypotheses to switch between all levels of abstraction. We found that use of the situation model as a bridge between the program and top-down model was common for all tasks.
  - As expertise increases, the number of hypotheses related switches decreases, indicating use of other strategies for comprehension. Experts also switch more often directly between the program and top-down models.
  - Engineers with little or some prior experience with the code switch most often between models after making an hypothesis.

#### 5.8.4 Switching Between Models – Action Generated

- Programmers switch between all three levels of abstraction regardless of task.
- Lack of expertise means programmers do not have the tools necessary to easily make connections from the domain to the code and vice versa. They must necessarily build their mental models at levels close to the code, switching mostly between the program and situation models in both directions. This confirms Pennington's [36] results.
- Domain only experts have the knowledge necessary to build a high-level representation but need the situation model as an intermediate domain in which to break known components in the top-down model into smaller components that are closer to code. They are also able to utilize the situation model as a bridge from the program model into the top-down model.
- Domain and language experts have both the domain and language expertise that allows them to make direct connections between program and top-down models without the help of an intermediate situation model.

- Subjects with *little* or no experience with the code spend their time building models at levels close to the code and by cross-referencing between program and situation models.
- Subjects with *some* accumulated knowledge are more likely to switch between all three models at a fairly even rate.
- *Significant* experience with a particular piece of code means programmers have reasonably complete mental representations at all levels. Therefore, switching will be goal-based and focused on areas of change. This is why, for example, when adding new functionality, connections between the new functionality at the top-down level and actual code location at the program model level will be predominant.

#### 5.8.5 Role of Code Size

- The level of abstraction must necessarily increase with the size of code to be understood. This is why, for example, the subject who was porting several clients from one operating system to another worked mainly at the top-down model level while the subject who looked at only the main procedure of his system worked predominantly at the program model level.

#### 5.8.6 Information Needs and Tool Capabilities

- Corrective maintenance and General understanding need more detailed information than other maintenance tasks.
- Domain concepts and connected model information are key information needs.
- Language only experts appear to need more program model information than domain experts due to their tendency to work mostly at the program model level. Language and domain experts have the knowledge that allows them to articulate the type of information they need. They also look for a variety of information at all levels of abstraction.
- Engineers with *little* and *some* accumulated knowledge need more information in general than those who have more experience with the code. Those with *significant*

accumulated knowledge already have nearly complete mental representations at all levels with the appropriate connections between. The information they require is directly related to task. They also have fewer information requirements.

## Chapter 6

### FUTURE DIRECTIONS

This chapter contains several suggestions for further work based on the most important results. Since the objective of this observation was theory building, all suggestions for further work revolve around experiments to validate this theory.

#### 6.1 Integrated Model Construction

Our experiments used subjects covering a wide set of tasks. There were several debugging tasks and a few enhancement, adaptive maintenance, and general understanding tasks. Each subject was working on different code with different goals in mind. While commonalities among this diverse group of subjects emerged, statements about differences due to tasks must remain hypothetical until further single task experiments can confirm them. Experiments using the same task and identical code during equivalent time periods will provide insight and either increase our confidence in our conjectures or refute them. The analysis techniques we used are valid, including applying our classification of actions, hypotheses, switching conditions, and processes to the resultant protocols.

We hypothesize that accumulated knowledge can affect the amount of effort put into building mental representations at specific levels. For example, someone who has worked with the code for a significant period of time will have a good representation at all levels of knowledge. Mental model construction is driven mostly by task. If the task involves tracking down and fixing defects, the program model may be the focus. On the other hand, porting entire programs from one operating system to another may require use of higher-level domain information which makes completing the task much more efficient.

The effect of accumulated knowledge on mental model building can be tested in a couple of ways. Controlled experiments need to be designed in which expertise (domain and language) and task is held constant while amount of experience with the code is

varied from “never before seen” to “highly skilled”. Another experiment holds expertise and amount of accumulated knowledge constant while the task is allowed to vary. These experiments would be difficult to run because it is not easy to find subjects with the same expertise. A pre-test will need to be developed to accurately measure amount of expertise. It might also require a significant amount of pre-work to bring some subjects up to the rank of “significant” accumulated knowledge and others to “some” accumulated knowledge. Accumulated knowledge classifications need to be much more rigorously defined in such an experiment.

Expertise also affects mental model construction. We found that when domain knowledge is lacking, programmers spent the majority of their time building program and situation models. When sufficient domain knowledge exists, programmers worked at all levels, with focus apparently determined by task. Again, experiments can be designed similar to those for testing the effect of accumulated knowledge, except that here expertise is allowed to vary while task and accumulated knowledge is held constant. Similar to experiments on accumulated knowledge, the difficulties will occur with classification of experts.

Our investigation was an observational field study designed for building theory and generating hypotheses rather than testing hypotheses. Therefore we could not determine how task, expertise, and accumulated knowledge interact with each other. Since there is good indication from our results that significant interactions exist, a controlled factorial experiment should be designed to discover them. Such an experiment would contain groups of subjects with all possible combinations of expertise and accumulated knowledge working on each task. For an experiment using 3 levels of expertise, 3 levels of accumulated knowledge, and 4 task levels, this would require 36 (4x3x3) groups of subjects. Interaction can then be determined through an analysis of variance applied to the results. This experiment is probably not realistic for large-scale software tasks similar to the tasks used in our study.

In general we saw less use of the situation model than the top-down or program models. One reason for this is the definition of the situation model itself which includes both information at the algorithmic level and real-world knowledge. We did not see many references to algorithmic information during the programming sessions. Another reason for this could be related to a different aspect of this study. A majority of our subjects

were working on systems level programs, e.g. networking or operating systems. These have very little connections to physical objects in the real world.

An observational study similar to ours on programmers trying to understand object-oriented programs could help to develop more theories about the situation model. Classes can model real-world objects and their behaviors more completely than straight procedural programming with algorithms. Since objects should be defined so they model objects in the real world, we expect to see greater use of the situation model for object oriented program understanding. The situation model is the level at which these objects live. These theories could then be tested with more controlled correlational experiments. To date, all object-oriented comprehension experiments reported in the literature concentrate on program development rather than maintenance. We suspect that for object-oriented comprehension, the situation model will be utilized more. Pennington et al. [35] and Lee and Pennington [28] found that object-oriented experts used a systematic, top-down design strategy, breaking the problem space down into objects and the methods used by those objects. By contrast, procedure experts used a more opportunistic approach in which design occurred by switching between different levels of abstraction.

## 6.2 Hypothesis Generation

This section suggests further work on the HK-Ratio and hypothesis types.

### HK-Ratio and Expertise

Whether or not the HK-Ratio is a good measure of expertise can be tested by designing a longitudinal study with a sufficient number of novice subjects. An initial HK-Ratio can be taken by giving each subject a task involving a large program and measuring the number of hypotheses and references to knowledge. Periodic measurements can be taken during the course of several sessions while they attempt to complete the task. If the HK-ratios consistently drop as expertise and accumulated knowledge increase, we can conclude the HK-ratio is a good indicator of expertise. An additional experiment would need to be designed, perhaps using the same subjects, in which a debugging task is assigned. If the HK-ratio suddenly rises due to an increase in the number of hypotheses generated, then our theory that corrective maintenance can increase the HK-ratio will be confirmed. The

major difficulty with this experimental design is that longitudinal studies take significant time to complete and it can be challenging to find subjects who do not have past related experience that may significantly affect the outcome.

### **Hypothesis Types**

Further work is required to verify the most important types of hypotheses. We found that hypotheses about domain concepts, functionality, and low-level information like control-flow and data structures were very important for our group of subjects. Additional studies should be designed and analyzed for these specific types. Since hypothesis type may be determined by expertise, accumulated knowledge, and task, several experiments need to be designed to understand the effect of each variable. A factorial experiment using groups of subjects at all levels of expertise, accumulated knowledge, and task will help to determine the interactions of each of the three factors. A specific experiment using a corrective maintenance task will provide insight into the specific needs during debugging.

The major difficulties in designing an experiment that tests for specific types of hypotheses will be identification of subjects with the appropriate levels of expertise and accumulated knowledge and the development of tasks that are complex enough to elicit a rich set of hypotheses that can be analyzed.

### **6.3 Switching Behavior**

To confirm our conjectures on switching behavior, controlled experiments in which two of the three factors (task, expertise, accumulated knowledge) are held constant and the third allowed to vary should be designed. Number of switches between models can be measured using a method similar to ours for identifying actions and sequences of actions using protocol analysis. To test interaction between factors, a factorial experiment with larger groups of subjects at all levels for each factor can be designed.

### **6.4 Information Needs & Tool Capabilities**

We found a variety of information needs based on task, expertise and accumulated knowledge. Further work can be done to verify our hypotheses by designing experiments

similar to those described above for hypotheses, switching behavior, and the integrated model.

Tool capabilities alone provide a large area of further work. Evaluating prototype tools that address the issue of connecting domain, situation, and program model information is probably the most important, since this seemed to be the most frequent type of information our subjects looked for during comprehension activities. For example, a browser that utilizes an intelligent search mechanism for finding information specific to the task and the current level of abstraction at which a subject is working would address this need. One can imagine the ability to highlight a variable in the program and be able to browse related text-book domain descriptions or design decisions via hyperlinks.

Another important capability is the ability to look at information that was previously browsed regardless of how far back it was originally seen. This could be supported through a history mechanism and bookmarks (the capability to designate a location that can be returned to directly with a minimum number of keystrokes). This helps the resolution of hypotheses. For example, partially understood information (in the form of a deferred hypothesis) is integrated into the mental model, later something triggers the recollection of the hypothesis. With bookmarking and/or history of browsed locations, resolution of the hypothesis can be assisted by being able to return to the information that triggered the hypothesis in the first place. Bookmarks and a history of browsed data can also help with remembering hypotheses. Of course, one must have been conscious of having made and deferred an hypothesis.

Our study was an observational field study designed to construct a theory of program comprehension. Many of the suggested experiments proposed are correlational studies in which causal inferences can be made by statistically analyzing data from more controlled observational studies. The distinction is made between correlational and hypothesis-testing experiments. Hypothesis testing implies strictly controlled experiments with planned manipulation for eliminating extraneous influences. Our theories are not yet to a point where such hypothesis-testing experiments can be designed. Instead, they should be refined through correlational experiments with more subjects, stricter time control and tracking, and tasks common to groups of subjects.



This is a challenge. First, because the subjects, by and large, have to be professional programmers. It has not been easy to find volunteers. Second, our observations utilized large-scale production code. More controlled experiments would require giving the same large-scale production code to larger groups of professionals. This will be difficult. Thus, we expect that some elements of our theory will either take a long time to validate or will never be validated through controlled experiments.

## 6.5 Conclusions

Program understanding is a key factor in software maintenance and evolution. This thesis reported on an experiment with industrial programmers to discover comprehension activities, processes, hypothesis behavior, and supporting information that programmers need when trying to understand production code.

While our conclusions need further experimental validations through future experiments (our sample is small) we hope that our results stimulate further such experiments to increase knowledge of large scale code cognition.

Our subjects worked on production software using modern procedural languages. Future experiments could investigate whether the results hold for cognition with programming paradigms like visual programming and object-oriented programming. Petre [38] found that graphical representations can have the same pitfalls as textual representations of programs, i.e. graphical programs are not necessarily more comprehensible and can potentially be more problematic. Moher et al. [32] confirmed these results using Petri Nets as the graphical components.

In an experiment designed to find differences between object-oriented design experts and procedural design experts, Pennington et al. [35] and Lee and Pennington [28] found that the object-oriented experts used a systematic, top-down design strategy. In contrast, the procedural experts used a more opportunistic approach in which design occurred at different levels of abstraction, often jumping between levels. For maintenance activities, the obvious question is whether the shift to the object-oriented paradigm causes understanding to occur in a more systematic, top-down fashion as well.

Future experiments need to focus on understanding how programmers understand large-scale code written in object-oriented languages such as C++ and Smalltalk as well

as visual programming languages like Visual C++ and Visual Basic. Understanding hypertext (or hypermedia) is another important aspect of software comprehension as this technology is important for providing needed information during code understanding. The potential to create cognitive overload is especially significant in this paradigm. Thüring et al. [51] provide a set of design principles for hypermedia which focus on increasing the coherence between information and decreasing cognitive overload.

Obviously, tools that help cognition for large-scale code will help engineers to be more efficient and effective during maintenance. The integrated model and its corresponding comprehension processes have identified the importance of the availability of information about the program at different levels of abstraction. *RIGI* by Wong et al. [68] is a Reverse-Engineering tool that provides multiple views of software at various levels of abstraction. It is built for large-scale software.

Some tools and methods support Top-Down understanding. For example, hypothesis generation is supported by Rajlich, Doran, & Gudla's [40] tool, TLES. Mapping program structure to system architecture is also an important top-down comprehension activity. Canfora et al.'s [10] method derives program structure charts from system architecture. Abstract data type (ADT) identification is possible using Cimitile, Tortorella, & Monro's [12] method. It identifies Abstract Data Types by isolating the modules that implement the ADT.

A variety of tools and methods can be found that promote construction of more than one model. For example, Bennett & Ward's [5] middle-out programming method supports both top-down and situation model development during software design through the use of a high-level domain-specific language. The most ambitious tool for program and situation model support is Rich & Waters' [41] Programmer's Apprentice. This tool supports the program model level by recognizing common programming plans in small programs and builds the situation model using these *cliches*. Another tool that develops both a program and situation model is the Centre for Software Maintenance at Durham University's Maintainer's Assistant [4, 66]. This system uses a formal transformation method in taking program code and transforming it into non-executable specifications. Sometimes comprehension support is coupled with help for re-engineering support: an example is Achee and Carver's tool [1]. It identifies and extracts objects from legacy code.

Many more tools exist for development of program models. Common tools include static code browsers, call graphs, file graphs, and data-flow analyzers. An unusual program model tool is Blazy & Facon's [6] SFAC. Program behavior is simplified by freezing input values of parameters.

While these tools are important developments in supporting software comprehension, we also need further experiments on large-scale code in a variety of paradigms to increase our knowledge of how maintenance engineers understand code. This should drive further development of maintenance environments.

## REFERENCES

- [1] B. L. Achee and D. L. Carver, Identification and Extraction of Objects from Legacy Code, In: *IEEE Aerospace Applications Conference 1995*, February 1995, Snowmass, CO, vol. 2, p. 181-190.
- [2] Beth Adelson, Problem Solving and the Development of Abstract Categories in Programming Languages, In: *Memory and Cognition*, 1981, Vol. 9(4), pp. 422-433.
- [3] Beth Adelson and Elliot Soloway, A Model of Software Design, In: *The Nature of Expertise*, M.Chi, R. Glaser, and M.Farr (Eds), ©1988, Lawrence Erlbaum Associates, Publishers, pp. 185-208.
- [4] K.H. Bennett, Understanding the Process of Software Maintenance, In: *2nd Workshop on Program Comprehension, WPC'93*, July, 1993, Capri, Italy, p. 2-5.
- [5] K.H. Bennett and M.P. Ward, Theory and Practice of Middle-Out Programming to Support Program Understanding, In: *3rd Workshop on Program Comprehension, WPC'93*, November, 1994, Washington, D.C., p. 168-175.
- [6] Sandrine Blazy and Philippe Facon, SFAC, a Tool for Program Comprehension by Specialization, In: *3rd Workshop on Program Comprehension, WPC'93*, November, 1994, Washington, D.C., p. 162-167.
- [7] Deborah A. Boehm-Davis, Robert W. Holt, and Alan C. Schultz, The Role of Program Structure in Software Maintenance, In: *International Journal of Man-Machine Studies*, 36(1992), pp. 21-63.
- [8] Ruven Brooks,, Towards a Theory of the Cognitive Processes in Computer Programming, In: *International Journal of Man-Machine Studies*, 9(1977), pp. 737-751.
- [9] Ruven Brooks, Towards a Theory of the Comprehension of Computer Programs, In: *International Journal of Man-Machine Studies*, 18(1983), pp. 543-554.
- [10] G. Canfora, A. De Lucia, G.A. Di Lucca, and A.R. Fasolino, Recovering the Architectural Design for Software Comprehension, In: *3rd Workshop on Program Comprehension, WPC'93*, November, 1994, Washington, D.C., p. 30-38.
- [11] Ned Chapin, Software Maintenance Life Cycle, In: *Conference on Software Maintenance*, 1988, pp. 6-13.

- [12] A. Cimitile, M. Tortorella, and M. Munro, Program Comprehension Through the Identification of Abstract Data Types, In: *3rd Workshop on Program Comprehension, WPC'93*, November, 1994, Washington, D.C., p. 12-19.
- [13] Program Comprehension Workshop – CSM-92, *Workshop notes, IEEE Computer Society*, Conference on Software Maintenance, November 9, 1992, Orlando, Florida.
- [14] B. Curtis, I. Forman, and R. Brooks, and E. Soloway and K. Ehrlich, Psychological Perspectives for Software Science, In: *Information Processing & Management*, Vol. 20, No. 12, pp. 81-96.
- [15] Simon P. Davies, Models and Theories of Programming Strategy, In: *International Journal of Man-Machine Studies*, 39(1993), pp. 237 - 267.
- [16] Simon P. Davies, The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior, In: *Cognitive Science*, Vol.15 No. 4, October - December, 1991, pp. 547 - 572.
- [17] Simon P. Davies, The Nature and Development of Programming Plans, In: *International Journal of Man-Machine Studies*, 32(1990), pp. 461 - 481.
- [18] Francoise Detienne and Elliot Soloway, An Empirically-Derived Control Structure for the Process of Program Understanding, In: *International Journal of Man-Machine Studies*, 33(1990), pp. 323-342.
- [19] Francoise Detienne, Program Understanding and Knowledge Organization: The Influence of Acquired Schemata, In: *Cognitive Ergonomics: Understanding, Learning, and Designing Human-Computer Interaction*, ©1990, Academic Press, pp. 245-256.
- [20] K. Anders Ericsson and Herbert A. Simon, *Protocol Analysis: Verbal Reports as Data*, Second Edition, ©1993, MIT Press, Cambridge, MA.
- [21] K. Anders Ericsson and Herbert A. Simon, Verbal Reports as Data, In: *Psychological Review*, Vol.87, No. 3, 1980, pp. 215-251.
- [22] Edward M. Gellenbeck and Curtis R. Cook, An Investigation of Procedure and Variable Names as Beacons during Program Comprehension, In: *Empirical Studies of Programmers:Fourth Workshop*, December, 1991,New Brunswick, NJ, pp. 65-81.
- [23] Edward M. Gellenbeck and Curtis R. Cook, Does Signaling Help Professional Programmers Read and Understand Computer Programs?, In: *Empirical Studies of Programmers:Fourth Workshop*, December, 1991,New Brunswick, NJ, pp. 82-98.
- [24] Raymonde Guindon, Herb Krasner, and Bill Curtis, Breakdowns and Processes During the Early Activities of Software Design by Professionals, In: *Empirical Studies of Programmers:Second Workshop*, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 65 - 82.

- [25] Raymonde Guindon, Knowledge Exploited by Experts during Software Systems Design, In: *International Journal of Man-Machine Studies*, 33(1990), pp. 279-182.
- [26] Walter Kintsch and Teun A. van Dijk, Toward a Model of Text Comprehension and Production, In: *Psychological Review*, 85(5), 1978, pp. 363 - 394.
- [27] Jurgen Koenemann and Scott P. Robertson, Expert Problem Solving Strategies for Program Comprehension, In: *Proceedings of the Human Factors in Computing Systems, CHI'91*, New Orleans, May 1991, pp. 125-130.
- [28] Adrienne Lee and Nancy Pennington, The Effects of Paradigm on Cognitive Activities in Design, In: *International Journal of Man-Machine Studies*, Vol. 40, 1994, pp.577-601.
- [29] Stanley Letovsky, Cognitive Processes in Program Comprehension, In: *Empirical Studies of Programmers*, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 58 - 79.
- [30] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway, Mental Models and Software Maintenance, In: *Empirical Studies of Programmers*, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 80 - 98.
- [31] Katherine B. McKeithen and Judith S. Reitman, Knowledge Organization and Skill Differences in Computer Programmers, In: *Cognitive Psychology*, 13(1981), pp.307-325.
- [32] Thomas G. Moher, David C. Mak, Brad Blumenthal, and Laura M. Leventhal, Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets, In: *Empirical Studies of Programmers:Fifth Workshop*, Eds. C.Cook, J. Scholtz, and J. Spohrer, ©1993, Ablex Publishing Corporation, pp. 137 - 161.
- [33] Paul W. Oman and Curtis R. Cook, The Book Paradigm for Improved Maintenance, In: *IEEE Software*, January 1990, pp. 39-45.
- [34] Paul W. Oman and Curtis R. Cook, "Typographic Style is More than Cosmetic", *CACM 33*, May 1990, pp. 506-520.
- [35] Nancy Pennington, Adrienne Y. Lee, and Bob Rehder, Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design, In: *Human-Computer Interaction*, Vol. 10, 1995, pp.171-226.
- [36] Nancy Pennington, Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs, In: *Cognitive Psychology*, 19(1987), pp.295-341.
- [37] Nancy Pennington, Comprehension Strategies in Programming, In: *Empirical Studies of Programmers:Second Workshop*, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 100 - 112.

- [38] Marian Petre, Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming, In: *Communications of the ACM*, Vol.38, No.6, June 1995, pp. 33 - 44.
- [39] Roger S. Pressman, *Software Engineering, A Practitioners Approach*, 2nd Edition, ©1987, McGraw-Hill Publishing Co.,Inc.
- [40] Vaclav Rajlich, James Doran, and Reddi T.S. Gudla, Layered Explanations of Software: A Methodology for Program Comprehension, In: *3rd Workshop on Program Comprehension, WPC'93*, November, 1994, Washington, D.C., p. 46-52.
- [41] Charles Rich and Richard C. Waters, The Programmer's Apprentice Project: A Research Overview, In: *Artificial Intelligence & Software Engineering*, Derek Partridge,(Ed), ©1991, Ablex Publishing Corp., pp. 535 - 550. Reprint of IEEE Computer, 21(11), pp. 10-25. (AI)
- [42] Robert S. Rist, Plans in Programming: Definition, Demonstration, and Development, In: *Empirical Studies of Programmers: 1st Workshop*, 1986, Washington, D.C., pp. 28-47.
- [43] B.A. Sheil, The Psychological Study of Programming, In: *ACM Computing Surveys*, March 1981, Vol. 13, pp. 101 - 120.
- [44] Ben Shneiderman, Exploratory Experiments in Programmer Behavior, In: *International Journal of Computer and Information Sciences*, Vol. 5, No. 2, 1976, pp. 123 - 143.
- [45] Ben Shneiderman and Richard Mayer, Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results, In: *International Journal of Computer and Information Sciences*, Vol.8, No.3, 1979, pg. 219-238.
- [46] Elliot Soloway, Kate Ehrlich, Jeffrey Bonar, and Judith Greenspan, What do Novices know about Programming?, In:*Directions in Human/Computer Interaction*, Albert Badre and Ben Shneiderman (Eds), ©1982, ALBEX Publishing Corp., pp. 27-54.
- [47] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich, Cognitive Strategies and Looping Constructs: An Empirical Study, In: *Communications of the ACM*, November 1983, 26(11), pp. 853-860.
- [48] Elliot Soloway and Kate Ehrlich, Empirical Studies of Programming Knowledge, In: *IEEE Transactions on Software Engineering*, September 1984, Vol. SE-10, No. 5, pp. 595-609.
- [49] Elliot Soloway, Beth Adelson, and Kate Ehrlich, Knowledge and Processes in the Comprehension of Computer Programs, In: *The Nature of Expertise* , Eds. M. Chi, R. Glaser, and M.Farr, ©1988, A Lawrence Erlbaum Associates, Publishers, pp. 129-152.

- [50] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert, Designing Documentation to Compensate for Delocalized Plans, In: *Communications of the ACM*, Vol. 31, No. 11, November 1988, pp. 1259-1267.
- [51] Manfred Thüring, Jörg Hannemann, and Jörg M. Haake, Hypermedia and Cognition: Designing for Comprehension, In: *Communications of the ACM*, Vol.38, No.8, August 1995, pp. 57 - 66.
- [52] Iris Vessey, On Matching Programmers' Chunks with Program Structures: An Empirical Investigation, In: *International Journal of Man-Machine Studies*, (1987)27, pp.65-89.
- [53] Iris Vessey, Expertise in Debugging Computer Programs: A Process Analysis, In: *International Journal of Man-Machine Studies*, (1985)23, pp.459-494.
- [54] A. von Mayrhauser and A. Vans, Identification of Dynamic Comprehension Processes During Large Scale Maintenance, In: *IEEE Transactions on Software Engineering*, Vol. 22, Number 6, 1996, pp. 424 - 437.
- [55] A. von Mayrhauser and A. Vans, On the Role of Hypotheses During Opportunistic Understanding While Porting Large Scale Code, In: *Proceedings of the 4th Workshop on Program Comprehension*, Berlin, Germany, March 29-31, 1996, pp. 68 - 77.
- [56] A. von Mayrhauser and A. Vans, On the Role of Program Understanding in Re-engineering Tasks, In: *Proceedings of the IEEE Aerospace Applications Conference*, Vol.2, Snowmass, Colorado, February 3-10, 1996, pp. 253 - 267.
- [57] A. von Mayrhauser and A. Vans, Program Understanding: Models and Experiments, In: *Advances in Computers*, Vol. 40, M.C. Yovits and M.V. Zelkowitz (Eds), ©1995, Academic Press, Inc., pp. 1 - 38.
- [58] A. von Mayrhauser and A. Vans, Program Comprehension During Software Maintenance and Evolution, In: *IEEE Computer*, Vol. 28, No. 8, August 1995, pp.44 - 55
- [59] A. von Mayrhauser and A. Vans, Industrial Experience with an Integrated Code Comprehension Model, In: *IEEE Software Engineering Journal*, September 1995, pp. 171-182.
- [60] A. von Mayrhauser and A. Vans, Comprehension Processes During Large Scale Maintenance, In: *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp.39-48.
- [61] A. von Mayrhauser and A. Vans, Dynamic Code Cognition Behaviors for Large Scale Code, In: *Proceedings of the 3rd Workshop on Program Comprehension*, Washington D.C., November 14-15, 1994, pp. 74 - 81.
- [62] A. von Mayrhauser and A. Vans, From Program Comprehension to Tool Requirements for an Industrial Environment, In: *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, pp. 78 -86, July 1993.



- [63] A. von Mayrhauser and A. Vans, From Code Understanding Needs to Reverse Engineering Tool Capabilities, In: *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE93)*, Singapore, July 1993, pp. 230 - 239.
- [64] A. von Mayrhauser and A. Vans, From Code Understanding Comprehension Model to Tool Capabilities, In: *Proceedings of the 5th International Conference on Computing and Information (ICCI'93)*, Sudbury, Canada, May 27-29, 1993, pp. 469 - 473.
- [65] A. Marie Vans, *Code Comprehension Model*, Masters Project, Colorado State University, 1992.
- [66] M. P. Ward, F.W. Calliss, & M. Munro, The Maintainer's Assistant In: *Proceedings of the Conference on Software Maintenance*, Miami, Florida, October 16-19,1989, pp. 307 - 315.
- [67] Susan Wiedenbeck, Processes in Computer Program Comprehension, In: *Empirical Studies of Programmers*, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 48 - 57.
- [68] K. Wong, S.R. Tilley, H.A. Müller, and M.D. Storey, Structural Redocumentation: A Case Study, In: *IEEE Software*, Vol.12, No.1, January 1995, pp. 46 - 54.

## Appendix A

### CLASSIFICATION OF MODEL COMPONENTS – TASK CODES

#### A.1 Top-Down Component Identification

##### OP1 - Gain high-level overview of program

Specific intentions to get a high level understanding. Comments indicating that only a high level understanding is sought. For example, studying the file structure of a program. Also includes comments about obtaining a high level understanding, for example, "I see how this program segment fits into the whole."

##### OP2 - Examine/search for program information

Comments indicating that particular documents, pieces of documents related to the program, or information within specific documents are sought and read. Documents associated with the program such as maintenance manuals, design documents, user manuals are included. However, search in code is considered part of the bottom up model.

##### OP3 - Generate or revise hypothesis about functionality

Anytime a comment about the program in high-level terms is unsure, assumed, or not verified the comment is classified as an hypothesis. For example, "there is an initialization routine which I assume is called from..." Key words and phrases include: "*Probably*", "*I guess*", "*I assume*", "*It could be*", "*I wonder*", "*I think*," "*I believe*", "*It seems*".

##### OP4 - Determine relevance of program segment

Comments indicating the programmer is trying to decide if the segment being looked at is relevant to the current understanding task. Also included are comments that indicate that this decision has already been made. For example, "I'm not going to worry about that now", or "we don't care about that".

**OP5 - Determine if this program segment needs detailed understanding**

Similar to OP4. May be no difference since no protocol segments have been classified as such.

**OP6 - Determine understanding strategy**

Comments regarding how the programmer will or would have approached understanding at a high level. For example "First I will read the Requirements Documents, then ....".

**OP7 - Investigate Oversight**

Comments about overlooking some aspect of the code, for example forgetting to add some functionality or a whole routine.

**OP8 - Failed Hypothesis**

Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are about the high level structure of the program. For example, "I was wrong, there are not just two libraries there's a whole multitude of libraries." See Top-Down knowledge below.

**OP9 - Mental Simulation**

Comments that indicate the programmer is simulating (control-flow or data flow) at a high-level, for example at the procedure level, not at the procedure statement level.

**OP11 - High-level change plan/Alternatives**

Descriptions of changes to code or alternatives being considered at a high level. For example, "I will add the changes to the make file and then delete the print function." Or, "I could copy some files and then remove the directory containing the print function, or just leave things as they stand now."

**OP12 - Observe Buggy Behavior**

Comments indicating that program execution behavior is being observed and that it is

not correct or expected. For example, "I got an error, normally this would not take this long if it had been doing shared library versions."

### **OP13 – Study/Initiate Program Behavior**

Descriptions of steps being taken to initiate program execution or that indicate that program execution is being observed. Comments describing the intention of executing commands on the command line or comments describing actual evidence that the program is executing.

### **OP14 – Compare Program Segments**

Comments describing some kind of comparison at the program/application level in terms of functionality. This does not include direct comparisons of code, for example, results of the diff command. Comments that indicate that some similar program should be/is being examined for reuse at any level, eg. design, specification, etc. is a candidate for this category. Also included are comments that include the series of steps necessary to compare program segments.

### **OP15 – Generate Questions**

Verbalized high level questions that are either immediately pursued or noted for later investigation. These questions exist at the specification, design, application level. For example, "Did the program install the correct application defaults?"

### **OP16 – Answer Question**

Verbalized answers to stated questions, either immediately preceding the verbalization of the question, or if it directly answers a question that can be found in the protocol at an earlier time. Following the example question under OP15 above, the statement "YUP! IT DID" would qualify as an OP16 statement.

### **OP17 – Chunk & Store knowledge**

Very high level statements indicating that some result has been realized. These typically involve statements about the program or document being studied AS A WHOLE. For

example, "So, this program seems to work."

#### **OP18 – Change Directions**

Comments that indicate that a different direction is going to be/is being followed. For example, "I am going to try something completely different, I am going to copy these files over here instead."

#### **OP20 – Generate task**

Any comments describing additional tasks that are discovered and need to be done some time in the future. These tasks involve high level tasks for instance, reading a particular document. Most tasks referred will occur sometime beyond the current programming session, unless a discovery is made and the task needs immediate attention.

#### **OP22 – Examine Results of Execution**

Comments describing the results of a program execution. These statements typically follow closely after a "watching execution" episode (OP12/13). These statements are concerned with high-level aspects such as files and directories. In addition, descriptions of actions taken in order to examine the results are included as are comments verifying the program behaved as expected. For example, " and it did install the program in that directory."

#### **Top-Down Knowledge**

Any comments indicating that knowledge acquired sometime in the past either before or during the programming session is being used in reasoning. Include comments about previous sessions, for example, reading documentation or examining file structure. Also included are statements using knowledge already acquired in the on-going program session, for example, knowledge that is chunked and stored. Top-down knowledge includes call-graph information, or an overall understanding how the program or a program segment works. The key is that knowledge comes from out of thin air to be used in program comprehension. This knowledge must be of the Top-Down type. Keys: References to code at a high level, e.g. style in which code was written, high-level language features like "modularity".

## A.2 Program Model Component Identification

### **SYS1 - Read introductory code comments**

Statements that are word-for-word reading aloud of comments in the code. This usually takes the form of a preceding statement such as “the comments say...” or “I’m reading the comments and it says...”

### **SYS2 - Determine next program segment to examine**

Any comment suggesting that a particular piece of code is sought after or comments indicating the programmer is looking for something to analyze. Key words and phrases include: “*I have to go find...*”, “*I’m looking for ...*”

### **SYS3 - Examine next module in sequence**

Comments reflecting code that is being read, word for word or in micro/macro-structure form or a description of what is being read using program model language. For example, the actual declaration of a variable.

### **SYS4 - Examine next module in control-flow**

Comments reflecting code that is being read, word for word or in micro/macro-structure form or a description of what is being read using program model language. The difference between this task and examining a module in sequence is that control-flow is followed. For example, “...then within PS-RECEIVE this..”, or “SET-SCREEN-MODIFIED is a sub-procedure..”

### **SYS5 - Examine Data structures and definitions**

Comments suggesting that a data structure is being examined. For example, “Because STACK is a fairly generic name. ...ok here’s the definition of the variable, so it’s *structure STATE*. So let me see if I can find a definition for *structure STATE*. And here it is.”

### **SYS6 - Slice on Data**

Comments that indicate that a data structure behavior is being examined over time. For example, “ STACK is initialized with the number 0 then procedure X pushes the value

5...”.

#### **SYS7 - Chunk & Store**

Summarizing comments about code structure and the programming language. For example, “So, DO-ERASE-ALL-UNPROTECTED is called based upon ERASE-ALL-UNPROTECTED command.” Keywords include *So*, *OK*, *Thus*.

#### **SYS8 - Generate or revise Hypothesis**

Anytime a comment about program code is unsure, assumed, or not verified the comment is classified as an hypothesis. Key words and phrases include: “*Probably*”, “*I guess*”, “*I assume*”, “*It could be*”, “*I wonder*”, “*I think*”, “*I believe*”, “*It seems*” .

#### **SYS9 - Construct Call Tree**

Comments about construction of where a particular procedure or variable is called from. For example, “I’m making a note of where SET-SCREEN-MODIFIED is referenced at. Or actually where it’s called from.”

#### **SYS10 - Determine understanding strategy**

Comments regarding how the programmer will or would have approached understanding. For example, “by the way as odd as it may seem that actually is a determining factor as to how I would have approached this problem. When I originally looked at SET-SCREEN-MODIFIED I found out that it is only called from six locations. So I can manually do this for a relatively short period of time. ... If there were 50 calls here I would not be doing this for the 50 calls.”

#### **SYS11 - Generate New Task**

Comments indicating additional tasks have been generated or discovered as a result of comprehension activities. For example, “Oh, I’m going to have to go back in later and add some code to.....” Most tasks referred will occur sometime beyond the current programming session, unless a discovery is made and the task needs immediate attention.

**SYS12 - Generate Question**

Verbalized questions about the code that are either immediately pursued or noted for later investigation. Specific questions as well as questions expressed as specific information needed. For example, "Let's see whether it runs over here on my HPUX workstation."

**SYS13 - Determine if looking at correct code**

Explicit comments about finding the correct code or not. For example, "I am not looking at the correct code here."

**SYS14 - Change Direction**

Comments indicating the programmer is taking a different direction. For example, a different section of code will be examined because the current code is not supplying the information sought after.

**SYS15 - Generate/Consider alternative code changes**

Specific comments indicating that code will be physically changed or the consideration of the effects of different code changes. For example, "Did the variable DATA get set?"

**SYS16 - Answer Question**

Comments that answer a specific question that was asked earlier about the code, either immediately preceding the verbalization of the question, or if it directly answers a question that can be found in the protocol at an earlier time. Following the example question under SYS15 above, the statement "YUP! IT DID" would qualify as an SYS16 statement. Also includes a simple Yes or No right after the question was asked.

**SYS17 - Add/Alter Code**

Comments describing the code change as it is happening, or code added in the immediate past.

**SYS18 - Determine location to set breakpoint**

Specific statements indicating the programmer knows exactly where to place a breakpoint



in the code or is searching for the exact location to put a breakpoint.

### **SYS19 - Failed Hypothesis**

Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are specifically about statements in the code. See Program Model Knowledge below.

### **SYS20 - Determine error/omitted code to be added**

Comments indicating that the programmer realizes an error has occurred or some aspect of the code is missing.

### **SYS21 - Mental Simulation**

Comments in program model language indicating the programmer is mentally simulating program behavior. For example, "But, since we're only processing the first two arguments, then we leave this procedure somewhere while exiting, SET-SCREEN-MODIFIED gets reset."

### **SYS22 - Compare Code between versions**

Comments describing some kind of comparison at the code level. This does include direct comparisons of code, for example, results of the *diff* command or RCS. Comments that indicate that some similar program should be/is being examined for reuse at the code level is a candidate for this category. Also included are comments that include the series of steps necessary to compare code segments.

### **SYS23 - Search for var definitions/use**

Comments indicating that a specific data structure is being sought for examination. For example, "I'm looking for an Xvar since it thinks that I went from 1 to len." This category does not include the actual data structure examination itself, this is covered by SYS5.

### **SYS24 - Search for Begin/End of block**

Statements indicating that the beginning and/or end of a physical block is being sought

after. For example, "This BEGIN goes with this END" or "I need to find the end of this while loop."

### **Program Model Knowledge**

Statements expressing information in the program model, for instance program language syntax. References to specific language constructs or variables, statements, and parameters. Any comments indicating that knowledge acquired sometime in the past either before or during the programming session is being used in reasoning. Include comments about previous sessions, for example, reading related sections of code. Also included are statements using knowledge already acquired in the on-going program session, for example, knowledge that is chunked and stored. " 'WITH OBJECT'. This is PASCAL here that I am dealing with. It is not a standard WITH statement", is a good example.

### **A.3 Situation Model Component Identification**

#### **SIT1 - Gain Situation Model knowledge**

Specific intentions to get a situation/application level understanding. Comments indicating that background information is sought. For example, studying a text book on the subject of the application area. These comments are usually in the form of reading aloud some documentation.

#### **SIT2 - Develop questions**

Questions or inquiries about the application area or situation. These questions are either immediately pursued or noted for later investigation. For example, "I wonder if they used the ISIS protocol for switcher communication?"

#### **SIT3 - Determine answers to questions**

Verbalized answers to stated questions, either immediately preceding the verbalization of the question, or if it directly answers a question that can be found in the protocol at an earlier time. Following the example question under SIT2 above, the statement "YUP! IT DID" would qualify as an SIT3 statement.

**SIT4 - Chunk & Store**

Comments summarizing actions in the situation model. These typically involve statements about related real-world concepts, for example, the concept the program is attempting to simulate. Key words and phrases include *"So, thus, OK"* .

**SIT5 - Determine relevance of situation knowledge**

Statements indicating a decision to understand/not understand a specific concept in more depth.

**SIT6 - Determine next info to be gained**

Comments indicating the programmer is looking for a clue to help determine what information to gain next that might help in the comprehension process.

**SIT7 - Generate or revise Hypothesis**

Any time a comment about the situation or functionality using situation model language is unsure, assumed, or not verified the comment is classified as an hypothesis. Key words and phrases include: *"Probably"*, *"I guess"*, *"I assume"*, *"It could be"*, *"I wonder"*, *"I think,"* *"I believe"*, *"It seems"* .

**SIT8 - Determine understanding strategy**

Comments regarding how the programmer will or would have approached understanding. This is similar to SYS10 and OP6, except statements address understanding in the situation model. For example, "First I need to understand the change in inventory calculations, then..."

**SIT9 - Determine if error exists (missing functionality)**

Comments indicating an error has been found using situation model language. For example, "The inventory reduction calculation is missing and the calculation for adding new inventory is incorrect."

**SIT10 - Failed Hypothesis**

Comments indicating that something that was thought to be the case has been disproved or abandoned. These comments are about the real world/situation level related to the program. For example, "I was wrong, there are not just two protocol strategies there's a whole multitude of them." See Situation Knowledge below.

#### **SIT11 - Mental Simulation**

Comments in situation model language indicating the programmer is mentally simulating program behavior. For example, "But, since we're only adding the new part inventory, then we need to create a new part number, then add the part number to the order database..."

#### **SIT12 - Compare functionality between two versions**

Comments describing some kind of comparison of functionality at the situation level. This does not include direct comparisons of code, for example, results of the *diff* command. Comments that indicate that some concept should be/is being examined for use is a candidate for this category.

#### **Situation Model Knowledge**

Statements expressing information in the real world. For example, statements about operating system functionality or application area. This knowledge is typically associated with what is accomplished not HOW it is accomplished. (This is program model knowledge.)

## Appendix B

### SUPPLEMENTAL DATA FOR RESULTS

The sections in this appendix parallel the outline of Chapter 5, i.e. Action analysis, hypotheses, switches, and information needs.

#### B.1 Actions

This section compliments information reported on the percentage of actions subjects spend in each level of the integrated model. The information is organized as a series of bar charts. Each chart represents a particular configuration of subjects for task, expertise, and accumulated knowledge. Subjects are identified on the x-axis labels together with the model-level. For example, **OPAD1** denotes subject **AD1**'s top-down actions. Model levels designated as **OP** are Top-Down actions, **SIT** are Situation model actions, and **SYS** are Program model actions.

##### B.1.1 Action Percentages By Task

Four task types were identified among the subjects: Adaptive, Corrective, Enhancement, and General Understanding. The General Understanding group contains one subject who was working on leveraging knowledge from an existing program written in Pascal for implementation in C. Since the majority of his actions were concerned mainly with understanding the Pascal program, we grouped him with the other General Understanding subjects.

Figure B.1 shows a surprising similarity between the two subjects working on adaptive maintenance and, more specifically, porting tasks. Both subjects had references to the top-down model 67% of the time. For program and situation models they were within 1% of each other. They also were domain experts and had significant experience with the code. We conjecture that adaptation for large-scale code requires domain knowledge due to the

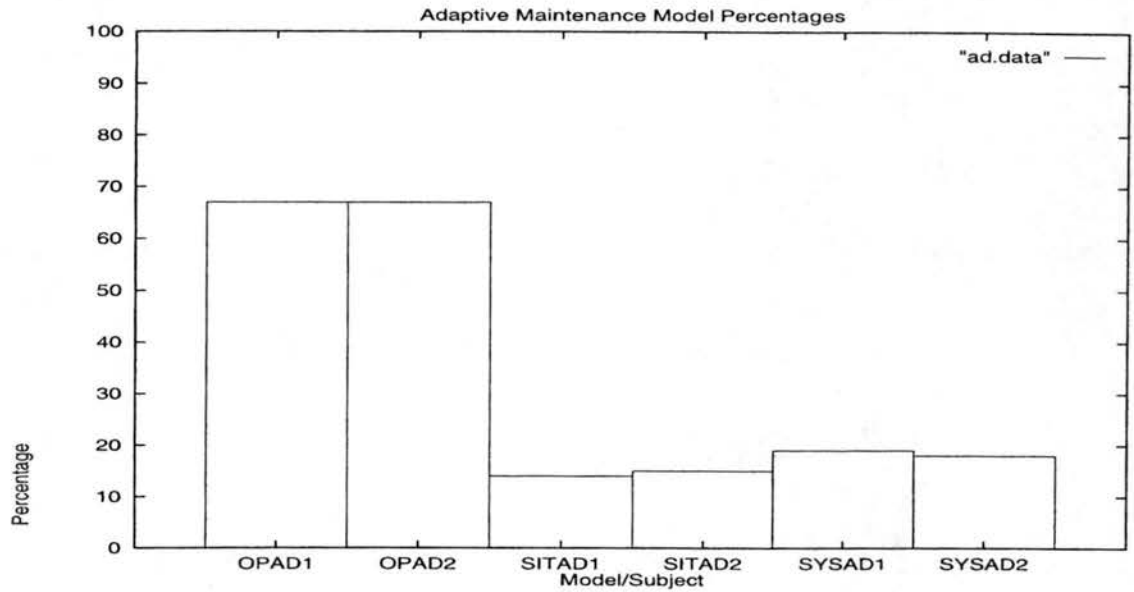


Figure B.1: Adaptive Maintenance Actions By Model (% of total actions, 2 Subjects)

size of the systems affected. For example, both subjects were porting code from one operating system to a very different system and if they had not had specific OS domain knowledge, they would have been unable to accomplish the amount of work in the two hour observation period. They kept referring to Top-Down knowledge as they looked at information associated with both systems and this knowledge helped them to focus on important aspects of the porting task.

The effect of debugging on the frequency of model references is not clear in Figure B.2. There are differences in expertise among the subjects and this may be interacting with the influence of the task. Subjects **C2** and **C3** have similar expertise (both are domain experts) and close to the same amount of accumulated knowledge. The percent of model references for these two subjects are closer to each other than between any of the other debugging subjects. Another hint that expertise may be interacting with the influence of task on model references is that subjects **C1** and **C4** have very different model reference behaviors. Subject **C1** is a language only expert while subject **C4** was a domain only expert. The frequency of top-down references for domain experts (who presumably have the knowledge necessary to break down the program into functional units) is high while the frequency of program model references is high for non-domain experts.

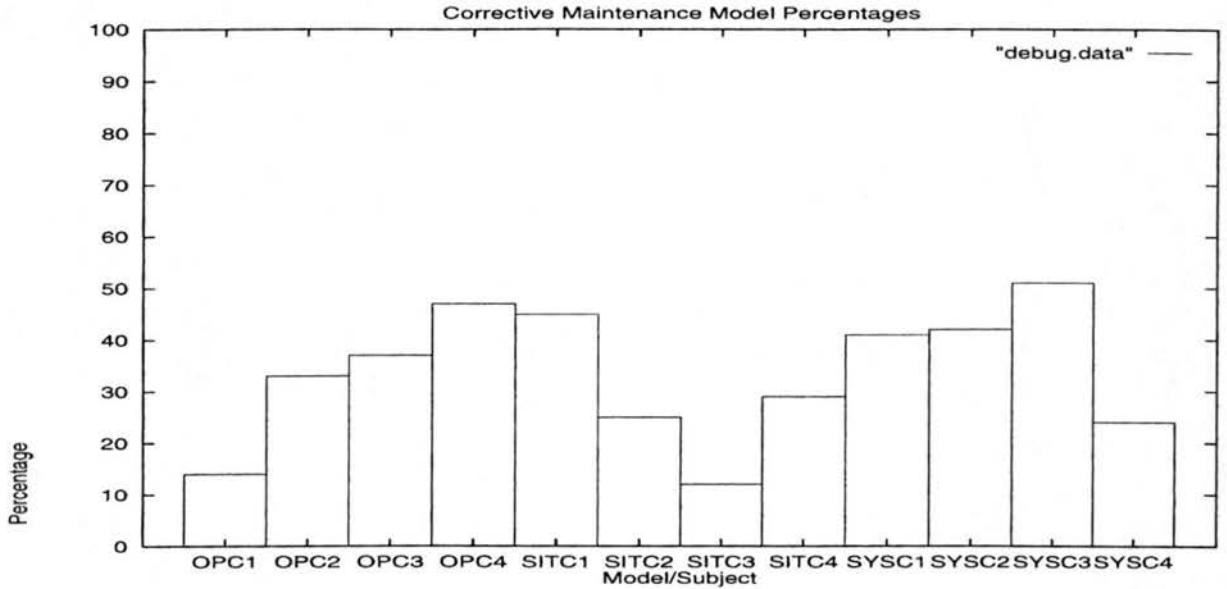


Figure B.2: Corrective Maintenance Actions By Model (% of total actions, 4 Subjects)

One trend is visible from the chart: for all the subjects working on debugging tasks, the number of references to the situation model is lower than that to either the top-down or program models. This appears to indicate that during debugging, connections are made between what the program is supposed to do and how it does it [52].

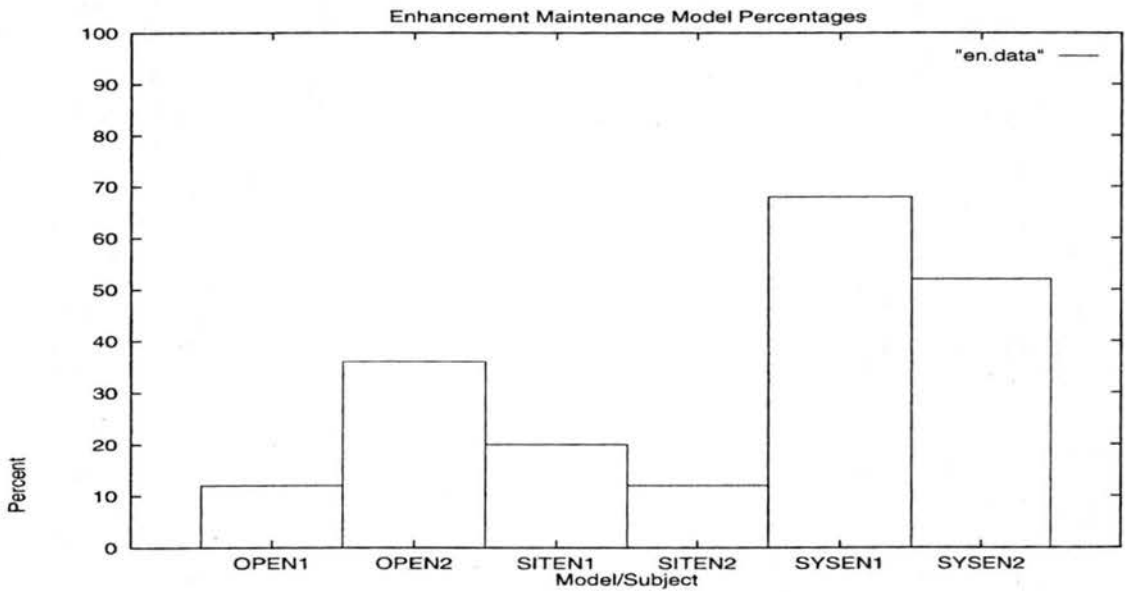


Figure B.3: Enhancement Maintenance Actions By Model (% of total actions, 2 Subjects)

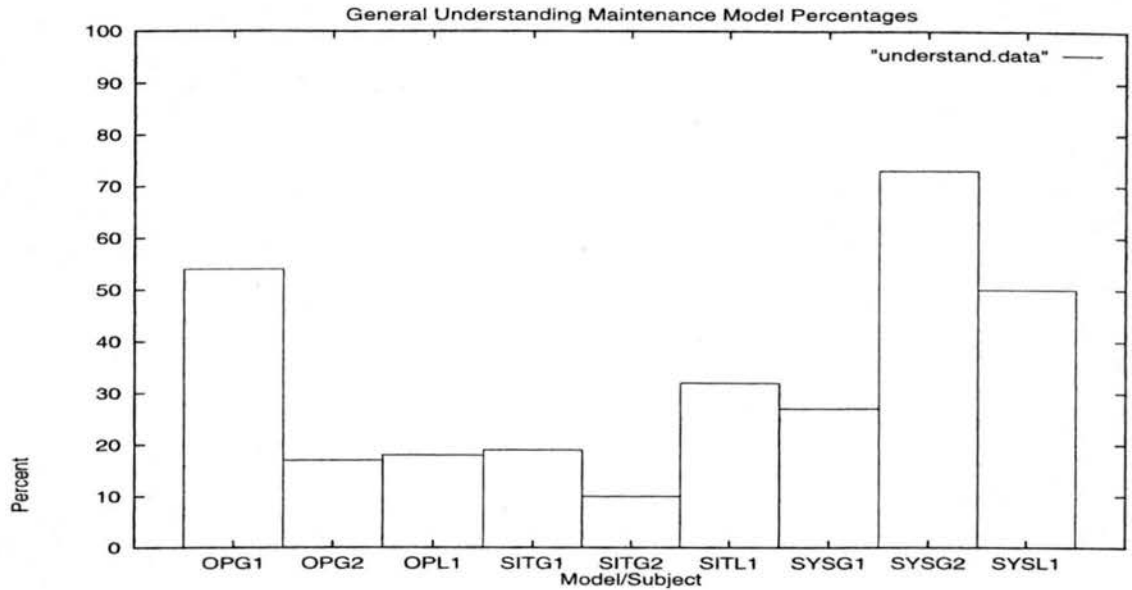


Figure B.4: General Understanding Actions By Model (% of total actions, 3 Subjects)

Similar to the debugging task, Figure B.3 does not clearly indicate a pattern common to enhancement tasks. This could be partially related to differing amounts of prior accumulated knowledge (**EN1** = significant; **EN2** = little) and contribute to the variance between them. There is a trend, however, of low top-down references and many more program model references. We might expect more top-down references and less program model references because both subjects were domain and language experts. **EN1** and **EN2** were working on adding specific functionality to the code. They were interested in locating and adding code to specific areas within the code, rather than spend time looking at unrelated pieces. Their task sequence was to find the appropriate location and concentrate on understanding the immediately surrounding code. Thus, it makes sense that they would spend more time building a program model than the domain model representation.

Figure B.4 is the last of the maintenance task graphs. This graph includes 3 subjects, one of whom had a goal of leveraging code (**L1**). Again, we see major differences between subjects. While **G1** and **G2** had similar amount of prior knowledge and expertise, we believe the differences between them is attributable to the overall strategies they employed during the understanding task. **G1** took an opportunistic approach in that he looked at Call Graphs and decided to examine modules based on what sounded interesting to him. **G2**, on the other hand, started with the main program of the system he was trying to



understand and looked at it line by line, systematically looking up variable definitions and function calls. The difference in L1's behavior may be attributable to the lack of both expertise and accumulated prior knowledge about the program he was trying to understand.

### B.1.2 Action Percentages By Expertise

The following bar charts depict graphically the clustering of relative frequencies by level of expertise. Three levels of expertise were identified among the subjects: Language Only, Domain Only, and Language & Domain.

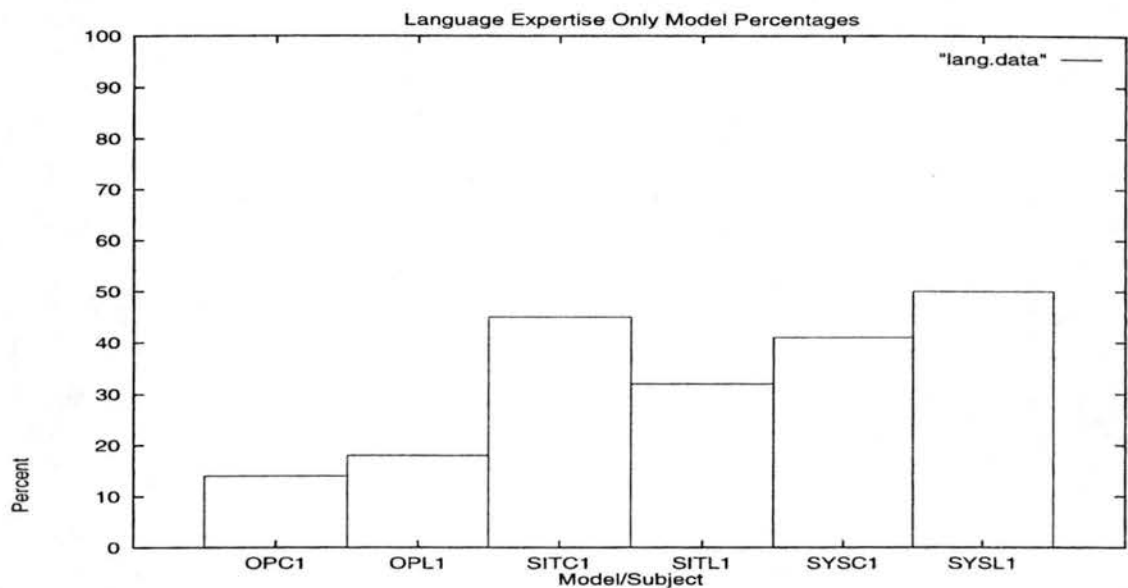


Figure B.5: Language Only Expertise By Model (% of total actions, 2 Subjects)

Results depicted in Figure B.5 (Language Only Expertise) support current theory [36] that people with little domain knowledge approach understanding primarily by building program and situation model mental representations. Between the two subjects, the maximum difference among same model references is only 14%, with 2 out of 3 under 10%.

Figure B.6 represents the distribution of actions for the domain only experts. Two subjects, C4 and AD2, show the effect of domain only knowledge. Both had considerably more top-down references than situation and program model references. The other two subjects, C2 and C3, showed the opposite pattern with more references to program model than top-down model. C2 and C3 were also close in percentage of references for each

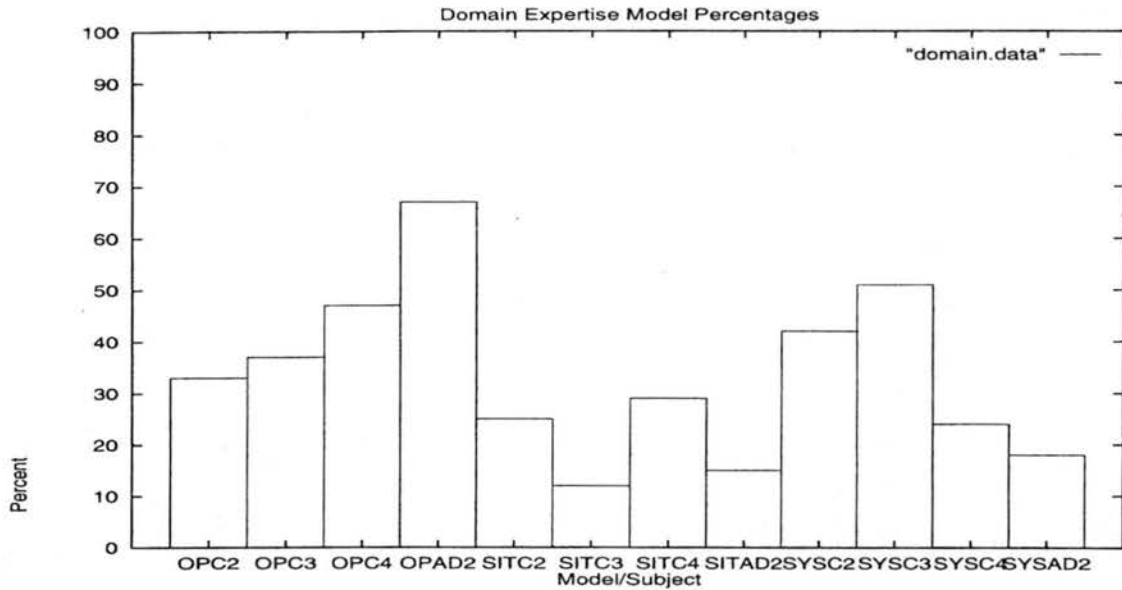


Figure B.6: Domain Only Expertise By Model (% of total actions, 4 Subjects)

model. We suspect this is because they were working on debugging tasks which are necessarily tied to program model representations. For example, if the goal is to fix a bug the expectation is that once the fault location is narrowed down, understanding what the code is actually doing is the most important goal. Both **C2** and **C3** were not only working on a debugging task, they also had very little knowledge about the code with which they were working.

Figure B.7 graphs the distribution of actions by model level for the domain and language experts. The following legend cross-references the code shown on the X-axis to model and subject:

Table B.1: Labels for Figure – Domain Only Expertise

<i>Top-Down Model Label</i>	<i>Situation Model Label</i>	<i>Program Model Label</i>
OA1 → Top-Down Model, AD1	STA1 → Situation Model, AD1	SSA1 → Program Model, AD1
OE1 → Top-Down Model, EN1	STE1 → Situation Model, EN1	SSE1 → Program Model, EN1
OE2 → Top-Down Model, EN2	STE2 → Situation Model, EN2	SSE2 → Program Model, EN2
OG1 → Top-Down Model, G1	STG1 → Situation Model, G1	SSG1 → Program Model, G1
OG2 → Top-Down Model, G2	STG2 → Situation Model, G2	SSG2 → Program Model, G2

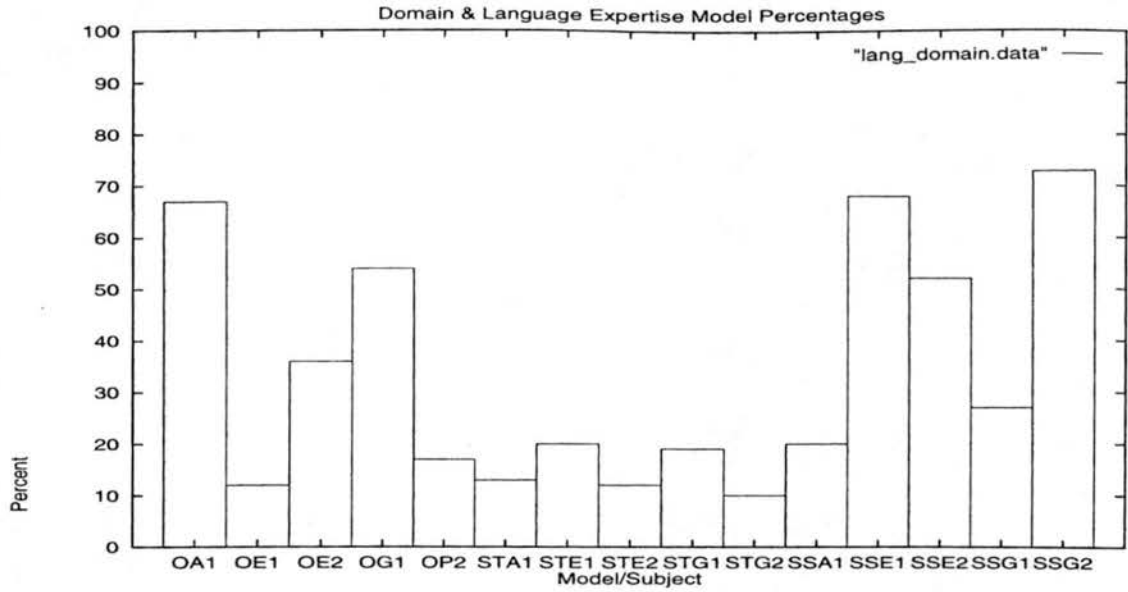


Figure B.7: Domain & Language Expertise By Model (% of total actions, 5 Subjects)

Figures B.7 shows no obvious trends for domain and language experts. Subjects **AD1** and **G1** are more similar to each other than to the rest of the subjects. This similarity is probably due to **AD1**'s porting task and **G1**'s opportunistic understanding strategy.

### B.1.3 Action Percentages By Accumulated Knowledge

Figures B.8, B.9, and B.10 cluster action distributions by level of accumulated knowledge. Three levels of accumulated knowledge were identified among the subjects: Little, Some, and Significant.

Figure B.8 represents this distribution of actions for subjects with very little prior experience with the software. Percentages among three of the subjects are very close. The last subject is different (within 25%). The only trend between models is that references to the situation model are lower than those to program and top-down models.

Similarly, Figure B.9 shows distribution of actions for subjects with some prior experience with the software. No obvious trends are visible from the graph.

Figure B.10 graphs relative frequencies of actions at all three model levels for subjects with significant prior experience with the software. Again, no obvious trends are visible from the graph. Accumulated knowledge may not be visible because expertise and task could be obscuring the effect.

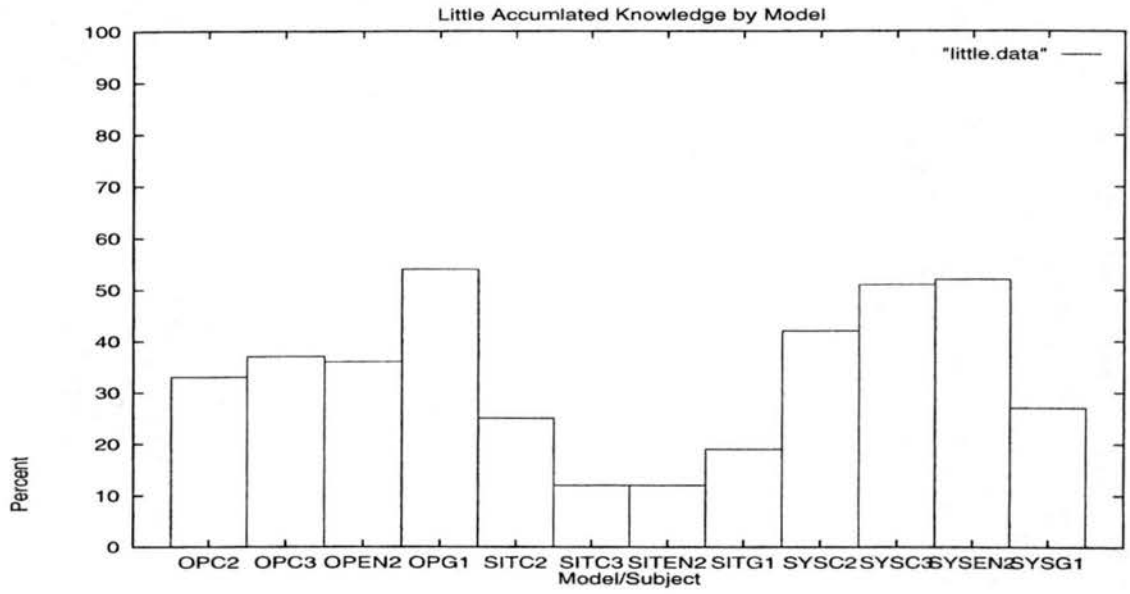


Figure B.8: Little Accumulated Knowledge By Model (% of total actions, 4 Subjects)

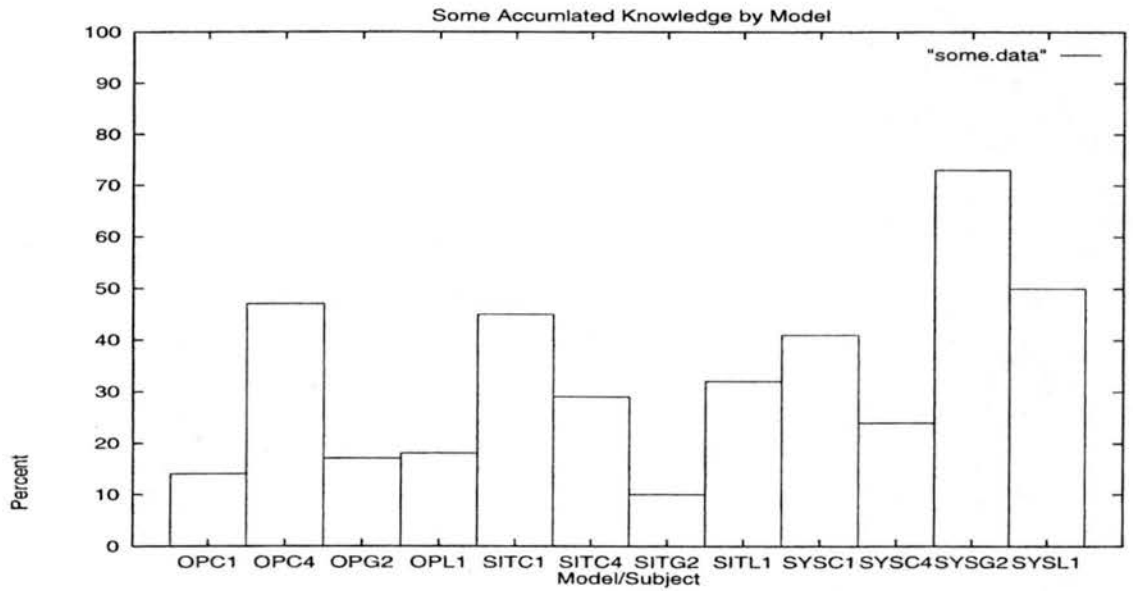


Figure B.9: Some Accumulated Knowledge By Model (% of total actions, 4 Subjects)

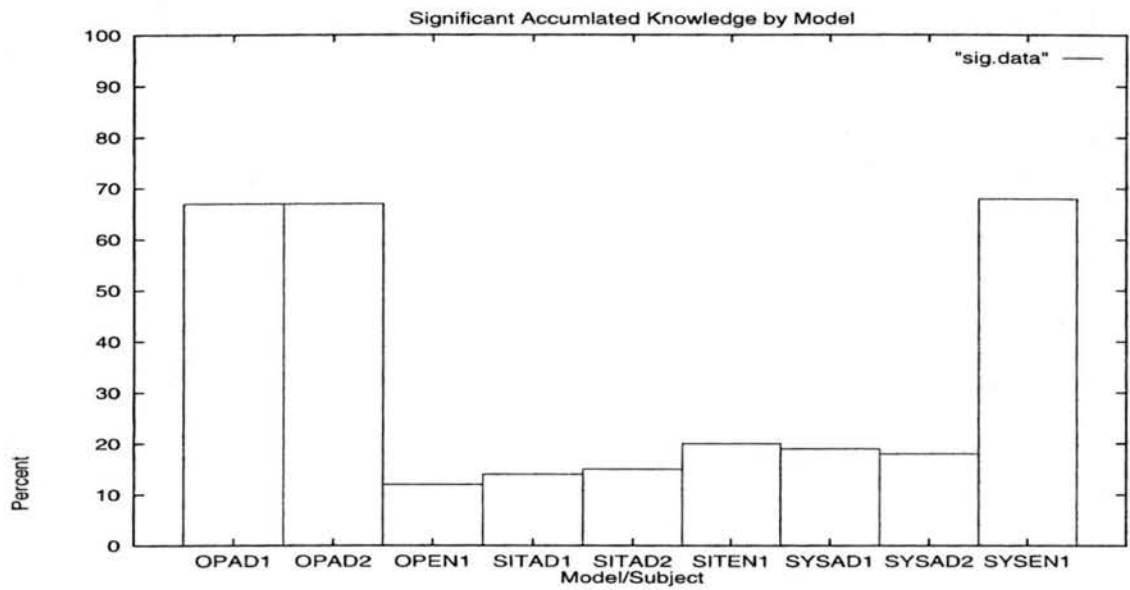


Figure B.10: Significant Accumulated Knowledge By Model (% of total actions, 3 Subjects)

## B.2 Hypotheses – Level 1

This section presents the detailed results of the hypothesis analysis for all subjects. It complements section 5.3. Result details are provided for the hypotheses types, hypothesis generated switches, and HK-Ratios. Table B.2 contains data on total top-down, situation, program, and combined situation and program model hypotheses for each individual subject. This table is helpful for determining overall counts and frequencies of hypotheses within each model.

Table B.2: Hypotheses by Model – Frequencies & Percentages

<i>Subject Code</i>	<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>	<i>Total Hypotheses</i>
<b>AD1</b>	23	14	3	40
<i>Adaptation</i>	57%	35%	8%	
<b>AD2</b>	42	6	3	51
<i>Adaptation</i>	82%	12%	6%	
Total	65	20	6	91
<i>Adaptation</i>	71%	22%	7%	
<b>C1</b>	9	17	24	50
<i>Corrective</i>	18%	34%	48%	
<b>C2</b>	18	18	12	48
<i>Corrective</i>	37%	37%	26%	
<b>C3</b>	26	7	18	51
<i>Corrective</i>	51%	14%	35%	
<b>C4</b>	24	22	11	57
<i>Corrective</i>	42%	39%	19%	
Total	77	64	65	206
<i>Corrective</i>	37%	31%	32%	
<b>EN1</b>	2	4	30	36
<i>Enhancement</i>	6%	11%	83%	
<b>EN2</b>	16	1	11	28
<i>Enhancement</i>	57%	4%	39%	
Total	18	5	41	64
<i>Enhancement</i>	28%	8%	64%	
<b>G1</b>	54	30	7	91
<i>Understand</i>	59%	33%	8%	
<b>G2</b>	13	7	40	60
<i>Understand</i>	22%	12%	66%	
<b>L1</b>	15	18	16	49
<i>Understand</i>	31%	37%	32%	
Total	82	55	63	200
<i>Understand</i>	41%	27%	32%	

## B.2.1 Model Frequencies by Task, Expertise, & Accumulated Knowledge

This section contains detailed results for hypothesis type analysis based on task, expertise, and accumulated knowledge. These are summarized and further interpreted in Chapter Five, section 5.3.1.

### B.2.1.1 Hypotheses Types By Task

A separate table for information on top-down (see Table B.3), program (see Table B.4), and situation model (see Table B.5) hypotheses for tasks is presented. The *Adapt* (Adaptive Maintenance) column contains data for two subjects, the *Corrective* (Corrective Maintenance) column reports on four subjects, the *Enhance* (Enhancement task) column shows data on two subjects, and the *Understand* (General Understanding) column represents data from three subjects.

#### Top-Down Hypotheses

Table B.3 contains the absolute and relative frequencies of top-down hypotheses for each type of task. These hypotheses are at a high level. For example, OPH8 (making hypotheses regarding whether the program runs correctly) is different than SYSH16 (code correctness/cause/location of error). In the first example, focus is on the entire program, while in the second, focus is on the specific piece of code. At the top-down level, the program may be run and behavior observed, whereas at the program model level a debugger may be used to step through specific code sections.

Table B.3: Hypothesis-Type Frequencies/Percentages – Top-Down Model, All Tasks

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Adapt</i>	<i>Absolute &amp; Percent of Corrective</i>	<i>Absolute &amp; Percent of Enhance</i>	<i>Absolute &amp; Percent of Understand</i>
OPH1	Domain Procedure function/Concepts	12 18%	11 14%	9 47%	33 40%
OPH2	Variable function/domain concepts	0 0%	2 3%	0 0%	5 6%
OPH3	Rules of discourse /Expectations	1 1%	1 1%	0 0%	5 6%
OPH4	I/O behavior	1 1%	0 0%	0 0%	4 5%
OPH5	Existence of installed (running) program	5 8%	0 0%	2 11%	1 1%
OPH6	Existence of specific functionality	4 6%	3 4%	0 0%	0 0%
OPH7	Number/type/existence/location of libraries	2 3%	5 7%	0 0%	0 0%
OPH8	Program functions correctly	9 14%	1 1%	0 0%	4 5%
OPH9	Permissions/Environment set correctly/Tool functionality	4 6%	11 14%	1 5%	0 0%
OPH10	Location to add functionality	0 0%	0 0%	3 21%	1 1%
OPH11	Comparison of functionality at high level	0 0%	2 3%	2 11%	0 0%
OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	6 9%	0 0%	0 0%	0 0%
OPH13	Number/type/location of file	3 5%	5 6%	0 0%	2 3%
OPH14	Available functionality	2 3%	2 3%	0 0%	10 12%
OPH15	Approaches/relative difficulty in making change	1 2%	0 0%	0 0%	0 0%
OPH16	Level & structure of code/scope	1 2%	11 14%	1 5%	15 18%
OPH17	Design Decisions/Modifications	0 0%	2 3%	0 0%	2 3%
OPH18	Location/Status/description/cause of error	14 22%	20 26%	0 0%	0 0%
OPH19	Current location	0 0%	1 1%	0 0%	0 0%



During the adaptive maintenance programming sessions, both subjects (AD1 and AD2) were adding code and trying to see if it ran correctly. It makes sense that many hypotheses would be generated around the question of why the newly added code wasn't functioning properly. The most common type of hypothesis observed during the sessions was in trying to determine location/status/description/ cause of errors (OPH18,22%). The second most common type of hypothesis was domain function/concepts (OPH1,18%). This may be due to the expertise (both were at least domain experts) and the amount of prior experience with the code (both had significant amounts of prior knowledge about the system they were trying to adapt). Because the systems were large, the only way to understand the impact of the changes was to approach understanding in an opportunistic manner. Making domain connections to program functions and procedures is one way to narrow the focus (and opportunistically determine which program components make sense to look at) when dealing with large systems. The third most frequent type of hypothesis is about whether the program functions correctly (OPH8,14%) (as opposed to figuring out what the error is).

Of all task types, adaptive maintenance contains the highest frequency of the OPH8 hypothesis type. This is reasonable since once a change is made, it is important to make sure the program functions correctly. However, we would also expect to see this behavior for both corrective and enhancement tasks. Yet, corrective maintenance contains only 1% and enhancement has no hypotheses of this type at all. In the case of the corrective maintenance sessions, none of the subjects reached a point in the task where they were able to identify exactly where the error was occurring and fix it. Given more than the two hours of the session, we would likely see more of these types of hypotheses. It is a similar case for the enhancement tasks. Both subjects had time only to find where to add the enhancement and add the needed code. Neither had much time to determine if the program functioned correctly after finishing the first two tasks.

For corrective maintenance, the majority of hypotheses concerned location/status/description/cause of error (OPH18,26%). This makes sense in light of their goal to fix a known defect. The other three most frequent types of hypotheses are evenly divided between domain procedure/function concepts (OPH1,14%), understanding the environment surrounding the system and understanding the tools they were using (e.g. the debugger)

(OPH9,14%), and level & structure of code (OPH16,14%). Code structure hypotheses are expected considering it is necessary to understand code structure when trying to pin-point bug location. We hypothesize that the reason domain concept hypotheses are frequent is because three of the four subjects were domain experts and used their grasp of the domain to direct their understanding of the bug.

For enhancing code, we find that the highest number of references are hypotheses about the connection of the domain to functionality in the program (OPH1,47%). It is difficult to determine whether this is related to the task or to the fact that both subjects were domain experts. It makes sense that because the focus of the task was to add functionality, a good grasp of the functionality already provided is necessary, but necessary at a high level rather than understanding each function at the code level. It also makes sense that because each subject was familiar with the domain, hypotheses could easily be generated about expected functionality. The second most frequent hypothesis type concerned the appropriate location to add the functionality (OPH10,21%). This is reasonable because the majority of both programming sessions was spent in locating the right place to add the new functionality. The other three tasks contained either none or very few of these types of hypotheses.

The highest percent of top-down hypotheses for General Understanding were hypotheses concerned with understanding the domain connections to functionality (OPH1,40%). Here it is difficult to tell if expertise or overall strategy is the major contributor. Two of the subjects in this category were domain experts. One of these two took a systematic approach by reading the code line by line while the other took an opportunistic approach by looking at code he thought might be interesting. The majority of the references to OPH1 came from the subject who approached understanding opportunistically (20 of the 33). Thus, it appears that strategy is the major influence in the number of references to domain-function related hypotheses. The second most common type of hypotheses are about level and structure of code (OPH16,18%). These hypotheses are concerned with structure at a high level, for example function-call structure. This is in contrast to program model level where statement execution order can be found. All 15 references to OPH16 were made by the domain expert who took an opportunistic approach to understanding. Again, strategy seems to play the dominant role, as this subject was mainly interested in

high level information about the system.

## Program Model Hypotheses

Table B.4: Hypothesis-Type Frequencies/Percentages – Program Model, All Tasks

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Adapt</i>	<i>Absolute &amp; Percent of Corrective</i>	<i>Absolute &amp; Percent of Enhance</i>	<i>Absolute &amp; Percent of Understand</i>
SYSH1	Variable function	1 17%	1 2%	0 0%	12 19%
SYSH2	Function/procedure function	0 0%	6 9%	4 10%	5 8%
SYSH3	Procedure/function call behavior	0 0%	0 0%	0 0%	1 2%
SYSH4	Variable structure	0 0%	7 11%	0 0%	10 16%
SYSH5	Location/type/existence of function call	0 0%	1 2%	3 7%	1 2%
SYSH6	Statement execution order/state	0 0%	15 23%	3 7%	7 11%
SYSH7	Variable value/defaults	1 17%	8 12%	8 20%	5 8%
SYSH8	(Non-)Existence of construct (var/code)	0 0%	3 5%	1 2%	0 0%
SYSH9	Variable/construct equivalency	0 0%	1 2%	0 0%	1 2%
SYSH10	Syntax meaning	0 0%	7 10%	0 0%	2 3%
SYSH11	Design decisions	0 0%	0 0%	0 0%	3 5%
SYSH12	Variable definition & it's location	0 0%	0 0%	0 0%	2 3%
SYSH13	Code block/procedure comparison	0 0%	2 3%	1 2%	1 2%
SYSH14	Code block function	0 0%	0 0%	0 0%	5 8%
SYSH15	Relevance of error in Makefile	1 17%	0 0%	0 0%	0 0%
SYSH16	Code correctness, cause/location of error	2 32%	10 15%	16 40%	1 1%
SYSH17	Changes made correctly	1 17%	0 0%	0 0%	1 1%
SYSH18	Location to add code/alternatives	0 0%	2 3%	5 12%	0 0%
SYSH19	Code block boundary location	0 0%	0 0%	0 0%	4 6%
SYSH20	Params/type defs in procedure call	0 0%	2 3%	0 0%	2 3%

Table B.4 contains the absolute and relative frequencies of program model hypotheses for each task type. These hypotheses are very low-level and tied closely to code.

Relative frequencies in the case of adaptive maintenance are somewhat misleading as both adaptive maintenance subjects made very few program model hypotheses overall. Therefore, we report results for this task by count, rather than relative frequency.

There were two hypotheses about code correctness or cause/location of error (SYSH16). This ties to the high number of hypotheses of the top-down hypothesis type, OPH18, location/status/ description/cause of error. Once a high level hypothesis about the error is made, hypotheses regarding the actual code can help direct focus to specific code segments. The other hypotheses types are variable function (SYSH1) and value (SYSH7), whether changes were made correctly (SYSH17), and hypotheses about make files (SYSH15).

The highest percentage of program model hypotheses for corrective maintenance subjects concerned statement execution order (SYSH6,23%). This appears to be related to task, because, when trying to locate a defect, understanding the order in which statements are executed (or could be executed, in the case of a branch) is important. The second most frequent type is code correctness (SYSH16,15%). Obviously, understanding whether the code is correct or not is an integral aspect of debugging. The third and fourth most important hypothesis types are variable value/defaults (SYSH7,12%) and variable structure (SYSH4,11%). Understanding variables and their values is essential when debugging code.

Interestingly enough, the highest percentage of hypotheses in the program model for enhancement activities is code correctness and location of error (SYSH16,40%). This is a very different result from that seen with the top-down model hypotheses. There were no references at all for top-down hypothesis type OPH18 (location/status/ description/cause of error). The reason may be due to the type of enhancement task subject **EN1** performed. 15 of the 16 references of this type were attributable to him. While he did not spend a lot of time compiling and running the program, he frequently recognized a potential or obvious error just by looking at code. When he saw a potential problem he proceeded to make hypotheses regarding the location of the source of the problem. The second most frequent types of hypotheses dealt with variable values (SYSH7,20%). Again, **EN1**

generated the majority of these hypotheses (7 out of 8). By contrast, **EN2** had very few references to the program model level, so it is no surprise that there is a lack of hypotheses at this level. (**EN1** had 37 program model hypotheses and **EN2** had 12). Some of **EN1**'s hypotheses about variable values were concerned with the same possible defects discussed above. Thus, it is reasonable to expect hypotheses about variables would follow from hypotheses concerning code correctness.

The majority of references for the understanding task concerns variable function (SYSH1,19%). Most of these references (5 out of 7) are attributable to subject **G2**, the subject who took the systematic approach to understanding code. The second and third most frequent types of hypotheses concern variable structure (SYSH4,16%) and statement execution order (SYSH6,11%), respectively. Again, one subject (**G2**) was responsible for the majority of the references. These top three types, taken together, constitute the major types of information at the program model level, with the exception of procedure structure and function. Thus, it is no surprise that the subject who took a systematic approach to understanding focused mostly on these three attributes of the program model.

### Situation Model Hypotheses

Table B.5 contains the absolute and relative frequencies of situation model hypotheses for each task type.

The highest percent of hypotheses for adaptive maintenance concerns program function (SITH8,40%). This appears to be related to the porting task on which one of the subjects (**AD1**) was working. 7 of the 8 references were attributable to him. The other adaptive maintenance subject (**AD2**) had the majority of the hypotheses references to functionality in the top-down model and only one hypothesis reference to functionality in the situation model. In contrast, **AD1** had the most hypothesis references to program function at the situation model level and no references to functionality at the top-down level. The porting task for **AD1** involved porting a series of X-clients to a different operating system. **AD1** needed to know how these clients were supposed to behave, independent of the OS. Thus, most of his hypotheses regarding functionality had to do with guesses about connections to real world behavior of each client. For example, one of the clients

Table B.5: Hypothesis-Type Frequencies/Percentages – Situation Model, All Tasks

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Adapt</i>	<i>Absolute &amp; Percent of Corrective</i>	<i>Absolute &amp; Percent of Enhance</i>	<i>Absolute &amp; Percent of Understand</i>
SITH1	Variable function	0 0%	7 11%	0 0%	16 29%
SITH2	Function/code block execution order/state	1 5%	9 14%	1 20%	7 13%
SITH3	Function/procedure function, call function	0 0%	25 39%	2 40%	17 31%
SITH4	Effect of running program	4 20%	1 2%	1 20%	0 0%
SITH5	Cause of buggy behavior	6 30%	6 9%	0 0%	0 0%
SITH6	Comparison of terms /acronyms/functionality	0 0%	4 6%	0 0%	1 2%
SITH7	Existence of function/algorithm/variable	1 5%	10 16%	1 20%	12 22%
SITH8	Program function	8 40%	2 3%	0 0%	2 3%
SITH9	Design Decisions	0 0%	0 0%	0 0%	0 0%

simulated a gas molecule in a heated box. At first, because he cued off the name of the program, *X-gas*, he thought perhaps it had something to do with gas stations.

The second most frequently referenced were to causes of buggy behavior (SITH5,30%). The majority of these were made by **AD2** (4 out of 6). **AD2**'s task was to test a port of a particular type of functionality using a prototype he had written. A lot of his time was devoted to setting up the new environment to test whether the new functionality would work. He ran into some problems with the environment and spent considerable effort trying to fix them. The third most frequent, effect of running program (SITH4,20%), were generated by **AD1**. He ran into very few problems during the task and usually successfully ran the client program on the OS he was porting to.

Hypotheses about functionality at the procedure level is most important for debugging (SITH3,39%). Our subjects seemed more concerned with functionality at the situation model level than at the program model level. Perhaps this is due to spending more time looking at statement execution order and variable values at the program model and abstracting functionality to the situation model. Hypotheses about variable and procedure functionality at the program model level is tied to language meaning; for example, "vari-

able 'i' is used as a looping mechanism". All corrective maintenance subjects were at least language experts, so the need to understand variable and procedure function is less important at the program level than understanding functionality at the algorithmic (or situation model) level. Using the same reasoning, the second most frequent hypothesis type, existence of function/algorithm/ variable (SITH7,16%), makes sense. During the understanding of a defect, it is important to determine whether some program construct that is critical to code correctness exists, for its absence may be a strong contributor to the defect.

It is difficult to draw conclusions about situation model hypotheses for the enhancement task because there were only five. Because of the low number of hypotheses, we report them by count rather than relative frequency. Two hypotheses were made relating to the function of a procedure (SITH3), one each to order or state of execution (SITH2), effect of running the program (SITH4), and existence of a function, algorithm or variable (SITH7). It is not clear whether these are typical situation model hypotheses for enhancement tasks. The data is too scarce.

The most frequent type of hypothesis for general understanding concern procedure function (SITH3,31%). Most of these were generated by **G1**, the subject who proceeded opportunistically. **G1** also had the majority of hypotheses about functionality at the domain level. For those functions he wanted to understand more thoroughly, he took time to understand them at the algorithmic level. We see the same behavior for the second and third most frequent hypothesis types – variable function (SITH1,29%) and the existence of expected functionality (SITH7,22%). In both cases, **G1** generated the majority of these references (11 out of 17, 8 out of 12, respectively).

#### **B.2.1.2 Hypotheses Types By Expertise**

This section contains a separate table for information on top-down (Table B.6), program (Table B.7), and situation model (Table B.8) hypotheses clustered by expertise level. In each table, the *Language* expertise column contains data for 2 subjects, the *Domain* expertise column reports on 4 subjects, and the *Domain & Language* expertise column represents data from 5 subjects.

## Top-Down Hypotheses

Table B.6: Hypothesis-Type Frequencies/Percentages – Top-Down Model, Expertise

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Language Expertise</i>	<i>Absolute &amp; Percent of Domain Expertise</i>	<i>Absolute &amp; Percent of Domain &amp; Lang Expertise</i>
OPH1	Domain Procedure function/Concepts	7 29%	22 20%	36 33%
OPH2	Variable function/ domain concepts	1 4%	1 1%	5 5%
OPH3	Rules of discourse /Expectations	1 4%	2 2%	4 4%
OPH4	I/O behavior	3 13%	0 0%	2 2%
OPH5	Existence of installed (running) program	0 0%	0 0%	8 7%
OPH6	Existence of specific functionality	0 0%	3 2%	4 4%
OPH7	Number/type/existence/ location of libraries	0 0%	5 5%	2 2%
OPH8	Program functions correctly	4 17%	6 5%	4 4%
OPH9	Permissions/Environment set correctly/Tool functionality	0 0%	13 12%	3 3%
OPH10	Location to add functionality	1 4%	0 0%	3 3%
OPH11	Comparison of function- ality at high level	0 0%	2 2%	2 2%
OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	0 0%	5 5%	1 0%
OPH13	Number/type/ location of file	1 4%	5 5%	4 4%
OPH14	Available functionality	0 0%	3 2%	11 10%
OPH15	Approaches/relative difficulty in making change	0 0%	0 0%	1 0%
OPH16	Level & structure of code/scope	5 21%	7 6%	16 15%
OPH17	Design Decisions/ Modifications	0 0%	2 2%	2 2%
OPH18	Location/Status/ description/cause of error	1 4%	33 30%	0 0%
OPH19	Current location	0 0%	1 1%	0 0%



It is difficult to draw conclusions about the effect of language expertise on top-down model hypothesis generation. There were too few subjects and too few total top-down hypotheses. These subjects had far more program model than top-down hypotheses (46 vs. 24). As reported in the action-type analysis section, the lack of top-down hypotheses in general may be due to a lack of domain expertise. The most frequent type of hypothesis, domain concepts tied to functionality (OPH1,29%), is not surprising as this was the most common top-down hypothesis type for all subjects. The second most frequent is more interesting because, although at the top-down level, level and structure of code (OPH16,21%) is more related to the program model than other top-down hypothesis types. If an engineer is not familiar with the domain, it seems reasonable that constructing the high-level structure of the program would be simpler given an understanding of the program model level. As previous research has shown [36], when experience in the domain is lacking, the program model tends to be built before connections to higher-level information can occur. Again, only two subjects made up this category and we expect that further experiments would show OPH16 to be a predominant type of hypothesis for language only experts.

For the domain experts, there are a considerable number of hypotheses in the top-down model. Percentages are spread among most of the hypothesis types (almost 80% of all types). The most important is OPH18 (30%, location/status/ description/cause of error). The most likely reason for this is that three of the four subjects were working on debugging tasks. The theory is that because they were all domain experts, hypotheses at the domain level could easily be formed. Looking at the table for program model hypotheses references, B.7, we see substantially fewer hypotheses in general. The total number of hypotheses in the top-down model are twice that of the program model (110 vs 51, respectively).

Domain and language experts have more domain concept hypotheses (OPH1,33%) than any other type of hypothesis. The subjects in this group include two general understanding, two enhancement, and an adaptive maintenance subject. It is tempting to say that because they were domain and language experts, they could easily generate hypotheses at the top-down level regarding high level functionality. However, two subjects, **G2** and **EN1** did not have nearly as many hypotheses in the top-down model as the

other three subjects. While **G2** had more OPH1 (Domain procedure/function concepts) hypotheses than any of the other top-down hypothesis types, **EN1** had none. For **G2** and **EN1** the lack of top-down hypotheses could be task related. Both of these subjects concentrated on understanding at levels close to program code. The theory is that expertise plays a role. However, it is not clear from this study that this is the case for language and domain experts. 15 of the 16 OPH16 (Level & structure code) hypotheses are attributable to **G1**. This is probably related to the opportunistic strategy he applied while understanding a large software system. His expertise allowed him to generate hypotheses on how he expected the program to be structured. Because he has the majority of the top-down hypotheses (84% of all generated), nothing can be concluded about effect of domain and language expertise on generating top-down hypotheses.

### **Program Model Hypotheses**

Table B.7 contains the absolute and relative frequencies of program model hypotheses clustered by level of expertise.

The highest percent of program model hypotheses for language only experts concern statement execution (SYSH6,32%). Pennington [36] found that when code is unfamiliar, programmers first build a mental representation that consists of an abstraction of the control-flow. When the domain is not familiar, connections to the top-down model can be difficult. Therefore, the easiest method for language only experts to understand code in an unfamiliar domain is to first build a control-flow abstraction before attempting connections to the domain. Syntax meaning (SYSH10,15%) is another logical hypothesis type for this group. Language only experts can use their expertise to form intelligent hypotheses about syntax during the program model building process. One of the third most important hypothesis types, code correctness (SYSH16,12%), is probably more influenced by task. Four of the five hypotheses about code correctness was generated by subject **C1** who was working on a debugging task.

For domain experts, program model hypothesis generation appears highly task influenced. Three of the four subjects worked on debugging tasks. In the case of hypotheses about code correctness (SYSH16,16%), this seems logical. The clue that program model hypotheses are task-related in this situation is that the one subject, **AD1**, who was not

explicitly working on a debugging task contributes only three program model hypotheses to this group of domain experts. As was the case with domain experts for top-down model hypotheses, we can theorize that the substantially fewer number of hypotheses in the program model may be attributable to domain expertise. We can't make any conclusions as to the types of hypotheses we expect to find within the program model since it appears that task is the driving factor for this group of experts.

The story for domain and language experts is different than that of domain only experts. The number of hypotheses domain-only-experts generated in the program model was small compared to the number generated in the top-down model. Domain and language experts made close to the same number of hypotheses in the program model as they did in the top-down model.

Looking more closely at the subjects in this group, **G2** contributes 63% of the total program model hypotheses and 48% of the total number of program model hypotheses (see Table 5.4). **G2** showed an unexpected pattern. This may be due to his use of a systematic strategy to understand code. If we disregard his contribution, since he is somewhat of an anomaly, the number of program model hypotheses is closer to what we would expect for language and domain experts, which is roughly half the number of top-down model hypotheses. Of course, we would have to disregard **G2**'s contribution to the top-down model as well, but this does not change the picture significantly as only 16% of the top-down model hypotheses were attributable to **G2**.

Again, disregarding **G2**, the top two most frequent types of hypotheses do not change. Code correctness (SYSH16) is still first, variable value/defaults (SYSH7) is second. The third most frequent type without **G2** is SYSH18 (location to add code). This is the same ranking as found in the enhancement task. With **G2**, the third most frequent task is SYSH1 (variable function). Subject **AD1** contributed only three program model hypotheses and **G1** contributed only 7 (out of 51 total without **G2**). The remaining two subjects, **EN1** and **EN2**, were responsible for the remaining 41 hypotheses. Thus, it appears that for program model hypotheses, these results are the same as those reported for hypothesis generation for program model building during enhancement. This is to say that we cannot draw conclusions about the effect of expertise for language and domain

experts (in the program model) because task seems to be the major influence for this group of subjects.

Table B.7: Hypothesis-Type Frequencies/Percentages – Program Model, Expertise

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Language Expertise</i>	<i>Absolute &amp; Percent of Domain Expertise</i>	<i>Absolute &amp; Percent of Domain &amp; Lang Expertise</i>
SYSH1	Variable function	2 5%	2 5%	10 11%
SYSH2	Function/procedure function	1 3%	6 14%	8 9%
SYSH3	Procedure/function call behavior	0 0%	0 0%	1 1%
SYSH4	Variable structure	3 7%	7 16%	7 8%
SYSH5	Location/type/ existence of func- tion call	0 0%	1 2%	4 5%
SYSH6	Statement execut- ion order/state	13 32%	4 9%	8 8%
SYSH7	Variable value/ defaults	2 5%	8 18%	12 13%
SYSH8	(Non-)Existence of construct (var/ code)	0 0%	3 7%	1 1%
SYSH9	Variable/construct equivalency	0 0%	1 2%	1 1%
SYSH10	Syntax meaning	6 15%	2 5%	1 1%
SYSH11	Design decisions	1 3%	0 0%	2 2%
SYSH12	Variable defini- tion & it's location	0 0%	0 0%	2 2%
SYSH13	Code block/ procedure compari- son	3 7%	0 0%	1 1%
SYSH14	Code block func- tion	2 4%	0 0%	3 3%
SYSH15	Relevance of error in Makefile	0 0%	0 0%	1 1%
SYSH16	Code correctness, cause/location of error	5 12%	7 16%	17 19%
SYSH17	Changes made cor- rectly	1 3%	0 0%	1 1%
SYSH18	Location to add code/alternatives	1 3%	1 2%	5 6%
SYSH19	Code block boundary location	0 0%	0 0%	4 5%
SYSH20	Params/type defs in procedure call	0 0%	2 4%	2 2%

## Situation Model Hypotheses

Table B.8 contains the absolute and relative frequencies of situation model hypotheses clustered by level of expertise.

Table B.8: Hypothesis-Type Frequencies/Percentages – Situation Model, Expertise

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Language Expertise</i>	<i>Absolute &amp; Percent of Domain Expertise</i>	<i>Absolute &amp; Percent of Domain &amp; Lang Expertise</i>
SITH1	Variable function	7 20%	5 9%	11 20%
SITH2	Function/code block execution order/state	5 14%	9 17%	4 7%
SITH3	Function/procedure function, call function	13 37%	17 32%	14 25%
SITH4	Effect of running program	0 0%	1 2%	5 9%
SITH5	Cause of buggy behavior	1 3%	9 17%	2 3%
SITH6	Comparison of terms /acronyms/function- ality	4 12%	1 2%	0 0%
SITH7	Existence of function/algorithm/ variable	5 14%	8 15%	11 20%
SITH8	Program function	0 0%	3 6%	9 16%
SITH9	Design Decisions	0 0%	0 0%	0 0%

The types of situation model hypotheses that are generated by language only experts concern abstractions of program code that are useful for understanding control-flow. We saw that statement execution order was the most frequent type of hypothesis for this group in the program model. It is logical that we should find this same group abstracting chunks of code into mental labels for use at the situation model level. These labels represent specific procedure/function and variable functionality. These two hypothesis types, SITH3 (37%) and SITH1 (20%), are the most frequent types of hypotheses generated by language only experts. A distant third are hypotheses concerning the existence of algorithms, functions, and variables (SITH7,14%). Tracking down expected program components is an activity that also helps to build a control-flow view of the program. This happens when some item is referenced by a specific code block and the engineer feels it necessary to understand it.

Generating hypotheses about procedure functionality at the situation model level (SITH3,32%) is the most frequent type of hypothesis for Domain only experts. As was the case with the top-down and situation models for this group, it is difficult to determine if this is an effect of expertise or task since all SITH3 hypotheses were generated by the three corrective maintenance subjects. The same holds for SITH2 (function state, 17%). For SITH5 (Cause of buggy behavior, 17%), five of the nine hypotheses were attributable to debugging tasks and four are attributable to **AD2**, the subject working on adding functionality to an existing program. In his case, he spent time testing a prototype he had written, and this part of his task is related to debugging. **AD2** spent a great deal of time trying to set up the environment in which to run the prototype. He had difficulty setting it up so the program would run correctly, so he therefore concentrated on determining the cause of the unexpected behavior he observed in the environment. The third most frequent hypothesis type, existence of functions, algorithms, and variables (SITH7,15%) is attributable only to the debugging subjects. To reiterate, it is not possible to determine the effect of expertise on situation model hypothesis generation because of the apparent influence of task. Obviously, what is required is another study in which domain experts work on a greater variety of tasks.

The greatest hypothesis frequency for domain and language experts concerns function-procedure functionality (SITH3,25%). 11 of the 14 hypotheses are attributable to **G1**, the subject who used an opportunistic strategy for the task. **G1** is also responsible for the majority of the second most frequent hypotheses types, variable function (SITH1,20%) and existence of function/algorithm/variable (SITH7,20%) (7 out of 11, and 8 out of 11, respectively). Interestingly, **G1** had 54 top-down model hypotheses and only 7 program model hypotheses. It appears that **G1** spent most of his time in the top-down model and used the situation model to make the majority of his connections to the program model. He rarely felt the need to go all the way to the program model level and was able to gather the information he needed at the algorithmic level (situation model). It is difficult to determine whether expertise or strategy is the dominant variable. One could argue that his expertise allowed him to make hypotheses that directed his progress through the task. On the other hand, his goal was to get a high-level view of the program. This could also be a major driver with expertise playing a secondary role.

For the remaining subjects, the frequencies change. The highest frequency for situation model hypotheses becomes SITH8 (program function, 26%), the second is SITH4 (effect of running program, 19%), and third is SITH1 (variable function, 15%). For SITH8, all remaining hypotheses were made by AD1; for SITH4, all hypotheses were generated by EN1 and AD1; for SITH1, all hypotheses were made by G2. So again, we can conclude very little about effect of expertise because the experts working on different tasks used dissimilar types of hypotheses. (EN2 generated only 1 situation model hypothesis during his programming session.)

### B.2.1.3 Hypotheses Types By Amount Of Accumulated Knowledge

This section reports results of hypothesis type analysis based on accumulated knowledge. The section contains a separate table for information on top-down (Table B.9), program (Table B.10), and situation model (Table B.11) hypotheses for accumulated knowledge groups. In each table, the column *Little Accum. Knowledge* contains data for four subjects. These subjects had either never seen the code prior to the session, they were familiar with the file structure, or had seen a call graph. Four subjects constitute the group with *Some* accumulated knowledge. This group had seen requirements and/or design documents and/or worked a little with the code prior to the programming session. The last group of subjects (*Significant Accum. Knowledge*) consists of three engineers who had previously completed maintenance tasks and/or worked with the code for several years.

#### Top-Down Hypotheses

Table B.9 contains the number of top-down hypotheses and the percent of top-down model hypotheses by amount of accumulated knowledge.

Subjects with *little* knowledge about the code spend the majority of their top-down hypotheses generation time on hypotheses about functionality (OPH1, 32%). A possible influence here is that all these subjects were at least domain experts. One strategy for understanding unfamiliar code is by generating high level hypotheses based on expectations and then investigating whether they hold. The second most frequent type (OPH16, 18%) is level and structure of code at a high level. This also makes sense for engineers

who have little experience with the program. Gaining an understanding of the structure is important for all tasks when the ultimate goal is to be able to maintain the code. It is not as easy to determine if the third most frequent (10%) hypothesis type (OPH14, hypotheses about available functionality) is related to lack of accumulated knowledge or domain expertise. On one hand, domain experts have the knowledge that allows them to generate hypotheses about domain concept connections to program functionality. On the other hand, none of the other accumulated knowledge groups found this type of hypotheses useful for their tasks. So it appears generating hypotheses about available functionality is unique to this group. It might be that there is an interaction between lack of accumulated knowledge and making hypotheses about functionality of the software. The domain expertise permits the generation of hypotheses about functionality they expect to find in programs within the domain. It is logical that, until more knowledge of the specific code exists, more hypotheses about expected functionality may occur.

The group with *some* experience working on the code generated more hypotheses about domain concepts (OPH1, 26%) than any other hypothesis type. This group consists of two corrective maintenance tasks and two general understanding tasks. Two subjects are language only experts, one is a domain only expert, and the other is a language and domain expert. The majority of the OPH1 hypotheses were generated by **L1** and **G2**, a language only expert and a language and domain expert, respectively. It appears that expertise does not play a significant role, but task may, since most of the OPH1 hypotheses were generated by subjects working on general understanding tasks. The two corrective maintenance subjects generated only three OPH1 hypotheses. However, it is possible that having some experience with the code affects the number and types of hypotheses generated. Understanding connections between domain and program functionality is important for building a top-down representation of the program. The frequency of OPH1 hypotheses for the group with *little* accumulated knowledge is higher than that for those with *some* accumulated knowledge. The frequency for the group with *significant* accumulated knowledge is lower than the other two. This trend can be explained by realizing that those with significant experience have already made most of the domain-functionality connections, the subjects with some accumulated knowledge have made some of the connections, and



the subjects with very little or no experience need to generate most of the connections. Thus, it is possible that the percentages reflect the effect of accumulated prior knowledge.

The second most commonly generated hypotheses for the group with *Some* accumulated knowledge, (OPH18, 20%), location/status/ description/cause of error was generated by the corrective maintenance subjects **C1** and **C4**. Most were generated by **C4** (11 out of 12). Task is probably the dominant influence but expertise also plays a part since **C4** is a domain only expert and **C1** is a language only expert. The third most common hypothesis, (OPH9, 15%) was generated entirely by **C4**.

It is not possible to make conclusions about effect of *significant* accumulated knowledge for top-down model hypothesis generation. The domain-functionality connection is important for the reasons stated above. Only two top-down model hypotheses were generated by subject **EN1** and the rest were generated by the two adaptive maintenance subjects, **AD1** and **AD2**. Thus, it is most likely that the results reported here are influenced mainly by task.

Table B.9: Hypothesis-Type Frequencies/Percentages – Top-Down Model, Accumulated Knowledge

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Little Accum. Knowledge</i>	<i>Absolute &amp; Percent of Some Accum. Knowledge</i>	<i>Absolute &amp; Percent of Significant Accum. Knowledge</i>
OPH1	Domain Procedure function/Concepts	37 32%	16 26%	12 18%
OPH2	Variable function/ domain concepts	4 4%	3 5%	0 0%
OPH3	Rules of discourse /Expectations	2 2%	4 7%	1 2%
OPH4	I/O behavior	0 0%	4 6%	1 2%
OPH5	Existence of installed (running) program	3 3%	0 0%	5 7%
OPH6	Existence of specific functionality	3 3%	0 0%	4 6%
OPH7	Number/type/existence/ location of libraries	5 4%	0 0%	2 3%
OPH8	Program functions correctly	0 0%	5 8%	9 13%
OPH9	Permissions/Environment set correctly/Tool functionality	2 2%	9 15%	5 7%
OPH10	Location to add functionality	3 3%	1 2%	0 0%
OPH11	Comparison of functionality at high level	4 4%	0 0%	0 0%
OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	0 0%	0 0%	6 9%
OPH13	Number/type/ location of file	5 4%	2 3%	3 4%
OPH14	Available functionality	12 10%	0 0%	2 3%
OPH15	Approaches/relative difficulty in making change	0 0%	0 0%	1 2%
OPH16	Level & structure of code/scope	21 18%	5 8%	2 3%
OPH17	Design Decisions/ Modifications	4 3%	0 0%	0 0%
OPH18	Location/Status/ description/cause of error	8 7%	12 20%	14 21%
OPH19	Current location	1 1%	0 0%	0 0%

## Program Model Hypotheses

Table B.10 contains the absolute and relative frequencies of program model hypotheses clustered by level of accumulated knowledge.

Subjects with *little* accumulated knowledge tend to generate hypotheses at the program model about procedure functionality (SYSH2, 17%). This ties into previously reported results on the top-down model hypothesis generation behavior for this group. All the subjects (**C2**, **C3**, **G1**, **EN2**) were at least domain experts and the hypothesis about procedure/function (SYSH2) at the program model level represent an attempt to make the connection between domain and program code. Also note that this group made fewer program model hypotheses per subject than the group with *some* accumulated knowledge (12.0 vs. 22.75 for *some*). The limited number of total program hypotheses makes it difficult to draw conclusions in this case. The high percent of hypotheses were generated mainly by **C2**, **C3**, and **EN2** with **C3** being responsible for the majority of the hypotheses (6 out of 7) to SYSH16 (Code correctness, 15%). As we said earlier when discussing the results for hypotheses generation by specific types of experts, we expect to see fewer program model hypotheses from domain experts. This is the case here.

Four subjects (**C1**, **C4**, **G2**, and **L1**) constitute the group with *some* accumulated knowledge. The most common program model hypothesis type relates to statement execution order/state (SYS6, 21%). The frequency may be related to accumulated knowledge, but it is difficult to confidently conclude this since the majority of this type were generated by one subject, **C1** (11 out of 19). However, it makes sense that generating hypotheses about program execution order (to develop a control-flow abstraction of the program) would be typical of engineers with some prior experience with the code. If the assumption is that this group had seen only requirements and design documents or had done some work with the code already, the control-flow abstraction would necessarily be incomplete. Thus, we would expect to see hypotheses that help to build the control-flow view. The same reasoning holds for understanding variable structure (SYSH4, 13%) and variable function (SYSH1, 10%). This group basically had at least one hypothesis for every type and they also had the highest average number of hypotheses per subject for the program model (22.75 per person). We conjecture that this group had enough prior experience to make lots of hypotheses about the actual code.

The last group contains three subjects, **AD1**, **AD2**, and **EN1**. It is not possible to make conjectures about effect of accumulated knowledge as it is likely that task masks all other influences. In the top-down hypotheses section, we reported that the adaptive maintenance subjects generated most of the top-down hypotheses. Here, adaptive maintenance subjects generated only six program model hypotheses. All remaining hypotheses were generated by **EN1**.

Table B.10: Hypothesis-Type Frequencies/Percentages – Program Model, Accumulated Knowledge

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Little Accum. Knowledge</i>	<i>Absolute &amp; Percent of Some Accum. Knowledge</i>	<i>Absolute &amp; Percent of Significant Accum. Knowledge</i>
SYSH1	Variable function	4 8%	9 10%	1 3%
SYSH2	Function/procedure function	8 17%	6 7%	1 3%
SYSH3	Procedure/function call behavior	0 0%	1 1%	0 0%
SYSH4	Variable structure	5 11%	12 13%	0 0%
SYSH5	Location/type/existence of function call	2 4%	1 1%	2 5%
SYSH6	Statement execution order/state	3 6%	19 21%	3 8%
SYSH7	Variable value/defaults	4 8%	10 11%	8 22%
SYSH8	(Non-)Existence of construct (var/code)	2 4%	1 1%	1 3%
SYSH9	Variable/construct equivalency	1 2%	1 1%	0 0%
SYSH10	Syntax meaning	2 4%	7 8%	0 0%
SYSH11	Design decisions	0 0%	3 3%	0 0%
SYSH12	Variable definition & it's location	1 2%	1 1%	0 0%
SYSH13	Code block/procedure comparison	1 2%	3 3%	0 0%
SYSH14	Code block function	1 2%	4 5%	0 0%
SYSH15	Relevance of error in Makefile	0 0%	0 0%	1 3%
SYSH16	Code correctness, cause/location of error	7 15%	5 5%	17 47%
SYSH17	Changes made correctly	0 0%	1 1%	1 3%
SYSH18	Location to add code/alternatives	5 11%	1 1%	1 3%
SYSH19	Code block boundary location	0 0%	4 5%	0 0%
SYSH20	Params/type defs in procedure call	2 4%	2 2%	0 0%

## Situation Model Hypotheses

Table B.11 contains the absolute and relative frequencies of situation model hypotheses clustered by level of accumulated knowledge.

Table B.11: Hypothesis-Type Frequencies/Percentages – Situation Model, Accumulated Knowledge

<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Absolute &amp; Percent of Little Accum. Knowledge</i>	<i>Absolute &amp; Percent of Some Accum. Knowledge</i>	<i>Absolute &amp; Percent of Significant Accum. Knowledge</i>
SITH1	Variable function	7 12%	16 25%	0 0%
SITH2	Function/code block execution order/state	9 16%	7 11%	2 8%
SITH3	Function/procedure function, call function	22 39%	21 33%	1 4%
SITH4	Effect of running program	0 0%	1 2%	5 22%
SITH5	Cause of buggy behavior	2 4%	4 6%	6 25%
SITH6	Comparison of terms /acronyms/functionality	1 2%	4 6%	0 0%
SITH7	Existence of function/algorithm/variable	13 23%	9 14%	2 8%
SITH8	Program function	2 4%	2 3%	8 33%
SITH9	Design Decisions	0 0%	0 0%	0 0%

For subjects with *little* accumulated knowledge, SITH3 (function/procedure functionality, 39%) is the most frequent type of hypotheses at the situation model level. This supports the theory that programmers who have little experience with the code focus on building connections between domain concepts and code. The situation model can be used as a bridge between the top-down and program models. Pennington [36, 37] found that when new code is encountered, programmers tend to build a control-flow view of the program by first building a program model and then abstracting to the situation model via a cross-referencing strategy, i.e. between program and situation model. Programmers often switch between all three models, including switches between situation and top-down models. This group of subjects with *little* accumulated knowledge switched between the top-down and situation models more frequently than between the program and situation models or between the top-down and program models. (See table B.19.)

Of course, expertise could be a factor. The subjects were at least domain experts. Their domain knowledge could be influencing a strategy that builds a mental representation, starting with domain expertise and proceeding by making connections to the code through the situation model. SITH7 (Existence of function/algorithm, 23%) was the second most common situation model hypothesis. Both expertise and amount of prior experience with the code could be playing a major role. Expertise helps to focus direction by generating hypotheses about the existence of program components and algorithms. Lack of experience drives the need to understand system composition.

Like subjects with *little* accumulated knowledge, subjects with *some* accumulated knowledge generated mostly hypotheses regarding procedure/function functionality (SITH3, 33%). The same reasoning used for subjects with little accumulated knowledge applies here. The percentage is slightly lower for those with some experience than those with little prior experience. This is logical since these subjects may have already answered questions those with little experience grapple with. However, their need to create a more complete mental representation means they must continue to make the connections between program and domain level. The same rationalization about function/algorithm existence (SITH7, 14%) holds for this group as did for the group of subjects with little experience. They are not at a point where they are familiar with most of the program components.

The second most frequent type for those with *Some* accumulated knowledge, variable function (SITH1, 25%), is interesting and could also be explained by the amount of prior experience. This group had the highest average number of situation model hypotheses (16.0 vs 14.0 for *little*, 8.0 for *significant*). If we look at the switching behavior for these subjects (Table B.20), we see that they switched between program and situation models more frequently than between situation and top-down models or program and top-down models. We surmise that because they had some experience with the code, they could generate intelligent hypotheses about the code. This group also had a higher average number of program model hypotheses than top-down model hypotheses.

Because the group with *significant* accumulated knowledge did not provide a spectrum of tasks, conclusions about accumulated knowledge are not possible as they may be more influenced by task rather than accumulated knowledge.

## B.2.2 Hypothesis Generated Switching Behavior

This section describes the results of an analysis on the effect of generating hypotheses on model switching. We report on hypotheses switching behavior for all subjects, by task, by expertise, and by accumulated knowledge. Section 5.3.2 contains a summary of the results presented in this section. In each hypothesis-generated switching table the row indicates the starting model and the column represents the ending model. Each table contains the total number of hypotheses (column labeled *Number of Hypotheses*) to highlight the difference between the total number of hypotheses-generated in each model and those that actually caused a switch. We also include the proportion of the switches for each type of model switch to illustrate the distribution of hypothesis generated switches across models.

### B.2.2.1 Hypotheses Switching – By Task

Tables B.12, B.13, B.14 and B.15 contain the number of switches caused by an hypothesis for each group of subjects working on a particular task type (adaptation, corrective maintenance, enhancement, and general understanding).

#### Adaptive Maintenance

Subjects working on adaptive maintenance generated a total of 91 hypotheses. 36 of them caused switches between model components. This represents 40% of all hypotheses generated.

Table B.12: Hypotheses Switching Behavior – Adaptation Task (AD1, AD2)

<i>TTI No. of Hypotheses: Adaptation</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
65	Top-Down	N/A	12 33%	3 8%
20	Situation	13 37%	N/A	2 6%
6	Program	3 8%	3 8%	N/A



The majority of the switches caused by hypotheses occurred between top-down and situation models. By contrast, if we look at Table B.28 (which reports all switches, not just those caused by hypotheses) we see a different behavior. This table shows that these subjects actually switched more often between program and top-down models than between top-down and situation models. For hypothesis generated switches, one explanation for the tendency toward top-down/situation model switching could be that task is the dominant influence. If we look back at Tables B.3 and B.5, we see that the majority of the top-down and situation model hypotheses were concerned with getting information about errors and whether the program they were trying to adapt ran correctly. In addition, since both subjects were domain experts, they could easily generate hypotheses at the top-down level and resolve them in the situation model or vice versa. This strategy allowed them to determine where things were going wrong with limited analysis at the program model level. Thus, the theory here is that generating and resolving hypotheses at levels higher than code can be a time saving strategy and efficient for tasks like porting code where code volume can be very high.

### Corrective Maintenance

Corrective maintenance subjects generated a total of 206 hypotheses. One hundred and eight of them were switches between model components. This represents 52% of the total hypotheses generated.

Table B.13: Hypotheses Switching Behavior – Corrective Task (C1,C2,C3,C4)

<i>Ttl. No. of Hypotheses: Corrective</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
77	Top-Down	N/A	19 18%	16 15%
64	Situation	23 21%	N/A	17 16%
65	Program	12 11%	21 19%	N/A

Table B.13 shows that hypotheses cause switches between models fairly evenly and often. Switches from the situation to the top-down model and from the program to the situation model happen at the highest frequency. This supports the theory that the situation model is used as a bridge from the program model to the top-down model, especially since

switching between the top-down and program models directly happen at the lowest rate. It is difficult to determine whether this is actually task related. However, we can say that because switching between all models occurs at a fairly even rate, corrective maintenance is a task for which making hypotheses at one level and using another level to help resolve them is important.

### Enhancement

Subjects doing an enhancement task generated 64 hypotheses. Only 25 of these hypotheses caused switches between model components. This represents 39% of the total hypotheses generated.

Table B.14: Hypotheses Switching Behavior – Enhancement Task (EN1,EN2)

<i>Ttl. No. of Hypotheses: Enhancement</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
18	Top-Down	N/A	0 0%	8 32%
5	Situation	1 4%	N/A	4 16%
41	Program	8 32%	4 16%	N/A

The switching behavior observed during enhancement is very different than for the rest of the tasks. Not only is the percent of hypotheses involved lower than the rest (39% vs. 40%, 49%, 52%), most of the switching occurs between the top-down and program models. Very few switches occurred between the situation and top-down model and there were substantially fewer switches between the program and situation models. This could be task related, but it is difficult to draw confident conclusions due to the limited number of switches. It is possible that because both subjects were working on enhancements, it makes more sense to jump between the program and top-down models. The table containing all switches for this task (Table B.28) shows that switches between top-down and program models take place at the highest rate. It could be that for enhancement tasks, it is less important to involve the situation model when generating hypotheses about domain/program functionality. Due to our small sample size, we can only theorize that we might see a greater degree of switching between program and top-down models.

## General Understanding

The general understanding subjects generated 200 hypotheses. Of these, 98 caused switches between model components. This represents 49% of the hypotheses generated by this group.

Table B.15: Hypotheses Switching Behavior – General Understanding Task (G1,G2,L1)

<i>Ttl. No. of Hypotheses: Understand</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
82	Top-Down	N/A	18 18%	12 12%
55	Situation	19 19%	N/A	18 18%
63	Program	13 13%	18 18%	N/A

The frequency of hypothesis caused switching for General Understanding is evenly distributed between top-down and situation models and between situation and program models. This is similar to the hypothesis switching behavior for corrective maintenance subjects. As for the corrective maintenance tasks, the situation model may be used as a bridge between top-down and program models. We can deduce this from the relatively few number of switches between the top-down and program models.

### B.2.2.2 Hypotheses Switching – By Expertise

Tables B.16, B.17, and B.18 contain the number of switches caused by an hypothesis (one for each level of expertise).

#### Language Only Expertise

The two language experts (C1 and L1) generated a total of 99 hypotheses. 67 of these generated switches between model components. This represents 68% of the hypotheses generated.

The highest percent of switching for language only experts occurred from the situation to the top-down models and from the situation to the program models. Also noteworthy was the number of switches from the top-down model to the situation model. We

Table B.16: Hypotheses Switching Behavior – Language Expertise (C1,L1)

<i>Ttl. No. of Hypotheses: Language</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
24	Top-Down	N/A	9 16%	7 12%
35	Situation	12 21%	N/A	11 19%
40	Program	5 9%	23 13%	N/A

conjecture that these subjects were using a cross-referencing strategy to build a program and top-down model using the situation model as a bridge. This is evident because the majority of switches occurred starting from the situation model. Lack of domain expertise is seen by the low frequencies of program to top-down model (and vice versa) switches. These subjects did not have the domain knowledge necessary to make direct connections between program and top-down mental representations based on expectations in the domain. As we will see below, the other two groups of subjects had domain knowledge and had a higher frequency of switches between program and top-down models. Another interesting result is that use of this strategy of making hypotheses at one level and switching to another model to help resolve them is important for language only experts. This group of subjects used 68% of their hypotheses in a switch to a different model. The other two groups used only 45% and 46% of their hypotheses in the same way.

### Domain Only Expertise

The four domain experts AD2, C2, C3, and C4 generated a total of 207 hypotheses. 95 caused switches between model components. This represents 46% of the total hypotheses generated.

Domain experts have fairly evenly distributed switches between all models, switching from the situation model to the top-down model slightly more often. Switching between top-down and program models occurred more often for this group than for the group of language experts. This is probably due their ability to make direct connections to the program model based on expectations at the top-down model. Also, the situation model can be used as a bridge between top-down and program models, which is analogous to decomposing the problem domain into components (intermediate domains) that are closer to

Table B.17: Hypotheses Switching Behavior – Domain Expertise (AD2,C2,C3,C4)

<i>Ttl. No. of Hypotheses: Domain</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
110	Top-Down	N/A	16 17%	16 17%
53	Situation	19 20%	N/A	14 15%
44	Program	13 14%	17 17%	N/A

the program. The opposite behavior can be seen through abstraction of program chunks.

### Domain and Language Expertise

Five domain and language experts, AD1, EN1, EN2, G1, and G2 generated a total of 255 hypotheses. 114 of these hypotheses generated switches between model components. This represents 45% of the hypotheses generated.

Table B.18: Hypotheses Switching Behavior – Language & Domain Expertise (AD1,EN1,EN2,G1,G2)

<i>Ttl. No. of Hypotheses: Language/Domain</i>	<i>From Model</i>	<i>To Model</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
108	Top-Down	N/A	24 21%	16 14%
56	Situation	25 22%	N/A	16 14%
91	Program	18 16%	15 13%	N/A

Domain and language experts also have fairly evenly distributed switches among models. Again there are enough switches between program and top-down models to demonstrate this group’s ability to make hypotheses based on expectations and resolving them at another level.

### B.2.2.3 Hypotheses Switching – By Accumulated Knowledge

Tables B.19, B.20, and B.22 report the number of switches caused by an hypothesis for each group of subjects who have different amounts of accumulated prior experience with the code.

### Little Accumulated Knowledge

The four subjects who had *little* accumulated knowledge, **C2**, **C3**, **EN2**, and **G1** generated a total of 218 hypotheses, of which 105 hypotheses generated switches between model components. This represents 48% of the hypotheses generated.

Table B.19: Hypotheses Switching Behavior – Little Accumulated Knowledge (C2,C3,EN2,G1)

Tot. No. of Hypotheses: Little	From Model	To Model		
		Top-Down Model	Situation Model	Program Model
114	Top-Down	N/A	21 20%	18 17%
56	Situation	24 23%	N/A	12 12%
48	Program	15 14%	15 14%	N/A

Subjects with *little* accumulated knowledge had more hypothesis generated switches between top-down and situation models than between any other models. The dominant effect, however, is probably domain expertise since all subjects were at least domain experts. Additionally, most of the hypothesis generated references between the top-down and situation models were generated by **G1**, who spent most of his time in the top-down and situation models. One theory is that since domain experts have the knowledge that allows them to generate hypotheses based on expectations, we would expect to see cross-referencing to and from the top-down model quite often. Also, task plays an important part. The one subject who generated most of the top-down to situation model switches was interested in understanding the system at a high level. Making hypotheses at the top-down model level and resolving them at the situation model level (and vice versa) seems a logical way to gain such an understanding.

Switching from the top-down model to the program model is also relatively frequent, however, the majority of these (12 out of 18) were generated by the two corrective maintenance subjects. The high frequency of top-down to program switches is probably due to the task since the goal here is to understand a defect and make direct ties to code. Thus, we cannot conclude much about the effect of little accumulated knowledge on hypothesis generated switching.

### Some Accumulated Knowledge

The four subjects with *some* accumulated knowledge, **C1**, **C4**, **G2**, and **L1**, generated a total of 216 hypotheses of which 110 hypothesis generated switches between model components. This represents 51% of the hypotheses generated.

Table B.20: Hypotheses Switching Behavior – Some Accumulated Knowledge (C1,C4,G2,L1)

Tot. No. of Hypotheses: Some	From Model	To Model		
		Top-Down Model	Situation Model	Program Model
61	Top-Down	N/A	16 14%	14 13%
64	Situation	19 17%	N/A	23 21%
91	Program	13 12%	25 23%	N/A

For subjects with *some* accumulated knowledge, switches generated by hypotheses involved the program and situation models at the highest frequency. As was the case with subjects with *little* experience, it is difficult to determine what is really affecting these numbers. **G2** was the subject who used the systematic strategy and therefore had no switches (caused by hypotheses) between the top-down and situation models. Thus, strategy can affect hypothesis switching behavior. By contrast, **C1**, **C4**, and **L1** employed an understanding strategy different from **G2** but similar to each other.

Table B.21: Hypotheses Switching Behavior – Some Accum. Knowledge (3 Subjects w/out G2)

From Model	To Model		
	Top-Down Model	Situation Model	Program Model
Top-Down	N/A	16 19%	9 11%
Situation	19 22%	N/A	16 19%
Program	7 8%	18 21%	N/A

Table B.21 looks more like the subjects who had domain only expertise. Switching between top-down and situation models or between situation and program models are

more frequent than switching between top-down and program models. Having some experience with the code allows them to make hypotheses about the code and use the situation model as a bridge to the top-down model. Rarely do they make hypotheses at the top-down level and switch to the program model or vice versa. (Refer to Figures C.17, C.23, and C.40 to show the actual sequence of hypothesis generated switches for each of these subjects.) In all cases, the situation model plays a central part in the journey from program to top-down models or from top-down to program models. We surmise that, with some experience, the subject has enough knowledge to use the cross-referencing strategy effectively. In fact, this group as a whole used 51% of all generated hypotheses for such a cross-referencing strategy. This is slightly higher than the percentage seen with subjects who had *little* accumulated knowledge (48%) and somewhat higher than subjects with *significant* accumulated knowledge (40%).

### Significant Accumulated Knowledge

The three subjects who had significant accumulated knowledge, **AD1**, **AD2**, and **EN1** generated a total of 127 hypotheses of which 51 generated switches between model components. This represents 40% of the hypotheses generated.

Table B.22: Hypotheses Switching Behavior – Significant Accumulated Knowledge (AD1,AD2,EN1)

Ttl. No. of Hypotheses: Significant	From Model	To Model		
		Top-Down Model	Situation Model	Program Model
67	Top-Down	N/A	12 23%	7 14%
24	Situation	13 25%	N/A	6 12%
36	Program	8 16%	5 10%	N/A

It is not possible to draw conclusions about the effect of *significant* accumulated knowledge on hypothesis generated switches. The majority of the switches were seen with one subject (**AD1**) who had 11 out of 13 switches from situation to top-down models and 10 out of 12 switches from the top-down to the situation model. **EN1** had no switches of this kind at all and **AD2** had only two each.



### B.2.3 HK-Ratio – Using Hypotheses and Knowledge to Indicate Expertise

Since the amount of accumulated knowledge seems to play an important part in code understanding, we wanted to determine if any patterns exist based on both the amount of accumulated knowledge and type of expertise. We also wanted to see how the HK-ratio corresponds to our classification of expertise. Tables B.23- B.27 contain matrices to illustrate HK-Ratios for program, situation, top-down, combined program and situation, and overall HK-ratios for accumulated knowledge and expertise. Each cell consists of the rank taken from Table 5.20 followed by the identifier for the subject who attained that rank.

Table B.23: Program Model: HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		4:C3	3:G1
		11:C2	8:EN2
<i>Some</i>	6:C1	6:C4	10:G2
	9:L1		
<i>Significant</i>		2:AD2	1:AD1
			5:EN1

In Table B.23 the subjects with the two lowest HK-ratios (ranks 1 & 2) in the program model possess either significant accumulated knowledge, domain only expertise, or domain and language expertise. The third lowest HK-ratio falls into the domain and language expertise column. The jump in HK-ratios is higher between the third and fourth ranked subject than between any two consecutive measures for the first three positions. Relatively speaking, subjects with the most domain and prior knowledge about the system have the lowest HK-ratio for program model construction.

On the other side of the scale, subjects with the highest HK-ratio fall into either the *some* or *little* accumulated knowledge categories, regardless of language or domain expertise. It seems reasonable that making program model hypotheses would be more frequent than use of prior knowledge if one is not familiar with the code.

Table B.24 shows the HK-ratio matrix for situation model building. HK-ratio ranks for situation model are very different than for the program model. The first two are both language and domain experts, however, they have very different amounts of accumulated

Table B.24: Situation Model: HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		4:C3 10:C2	11:G1 2:EN2
<i>Some</i>	5:C1 6:L1	8:C4	9:G2
<i>Significant</i>		3:AD2	7:AD1 1:EN1

knowledge. For situation model HK-ratios, ranking appears to be more related to task than expertise.

Table B.25: Top-Down Model: HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		5:C3 6:C2	11:G1 3:EN2
<i>Some</i>	9:C1 8:L1	4:C4	10:G2
<i>Significant</i>		7:AD2	2:AD1 1:EN1

Table B.25 contains the rankings for top-down model HK-ratios. Similar to the program model matrix, the first two (**EN1** and **AD1**) have significant accumulated knowledge and they are domain and language experts. The third lowest rank (**EN2**) is also a language and domain expert. It makes sense to say that the first two have the lowest HK-ratios because they both had significant accumulated knowledge and they were language and domain experts. However, because the first and third ranked subjects were working on enhancement tasks, the HK-ratios may also be influenced by task.

Ranks 4, 5, and 6 are all domain only experts and have either *little* or *some* accumulated knowledge. They were also all working on corrective tasks. Thus, these HK-ratios may be affected by task. The 8th and 9th ranked subjects are both language only experts and had *some* accumulated knowledge. It is reasonable to assume that both lack of domain knowledge and only *some* accumulated knowledge would result in a low level of expertise in the top-down model. HK-ratios for **L1** and **C1** were 1.0 and 1.5, respectively. This is high compared to .05 for **EN1**, the expert with the lowest HK-ratio. Finally, ranks 10 and 11 are both language and domain experts and have either *little* or *some* accumulated

knowledge. Since they were both General Understanding subjects, we conjecture that their HK-ratios were mostly influenced by task.

Table B.26: Combined Program & Situation Model: HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		4:C3 11:C2	9:G1 5:EN2
<i>Some</i>	6:C1 8:L1	7:C4	10:G2
<i>Significant</i>		1:AD2	2:AD1 3:EN1

We were interested in determining if rankings changed when we combined program and situation model HK-ratios, since their combination approximates Pennington's bottom up program comprehension model. Table B.26 has the rankings for the combined program and situation model HK-ratios. This matrix contains definite clusters of subjects whose HK-ratios seem related to expertise and accumulated knowledge. The first three ranked subjects all have significant accumulated knowledge and have either domain only or domain and language expertise. Ranks 6, 7, and 8 all have *some* accumulated knowledge and either language or domain only expertise. Subjects ranked 9, 10, and 11 had minimal accumulated knowledge and were either domain or domain and language experts. This last group contains both General Understanding subjects and as before, we believe this to be mostly related to that task.

Table B.27: Overall HK-Ratio Rankings

<i>Accumulated Knowledge</i>	<i>Language Expertise</i>	<i>Domain Expertise</i>	<i>Language &amp; Domain Expertise</i>
<i>Little</i>		5:C3 9:C2	11:G1 3:EN2
<i>Some</i>	7:C1 8:L1	6:C4	10:G2
<i>Significant</i>		4:AD2	2:AD1 1:EN1

Table B.27 shows the overall rankings for combined top-down, situation, and program models. We see clusters similar to the combined program and situation model and the top-down model HK-ratios. Those with the most accumulated knowledge and highest level of expertise have the lowest HK-ratio, while those with little or some accumulated knowledge

and language only expertise have higher rankings. Again, the highest HK-ratios were seen with the two General Understanding subjects and is probably more influenced by this task rather than expertise. It is interesting to note, however, that both subjects had either little or some accumulated knowledge, which could also be raising the HK-ratio.

### **B.3 The Role of Model Components in the Integrated Model – Level 1**

This section reports on detailed analysis of results on switching (whether due to hypotheses or other actions) between integrated model components. Chapter Five, section 5.4 contains a summary of these results.

#### **B.3.1 Switches Between Models**

Switches are defined as a change in focus from one level of abstraction to another. Switching can occur between the top-down, program, or situation models. Previously, we described switches caused only by hypotheses. However, any action can trigger a switch to another level of abstraction.

This section contains model switching information organized in three different ways: task, expertise, and accumulated knowledge. Each table contains the number of switches for each group of subjects. The row indicates the starting model and the column represents the ending model. We also include the proportion of the switches for each type to illustrate the distribution of switches across models.

##### **B.3.1.1 Switches Between Models – Totals By Tasks**

Table B.28 reports model switching behavior clustered by type of task.

#### **Adaptive Maintenance**

Switching between top-down and program models occurs (in either direction) most frequently for adaptive maintenance subjects. Switching between top-down and situation models occurs at a slightly lower rate. Switching between program and situation models happens at a much lower rate. Both subjects were working on a porting task. One subject, **AD1** had even higher percentages (see Table 5.22) for top-down - program model and top-down - situation model switches as well as lower program - situation model switches. In general, adaptive maintenance tasks like porting can involve large amounts

Table B.28: Action Switches – Absolute & Percent of Total Switches by Task

Task	Model	Model Switches – All Subjects		
		Top-Down Model	Situation Model	Program Model
Adaptive (2 subjects) (Total switches = 221)	Top-Down	N/A	42 19%	48 22%
	Situation	43 19%	N/A	20 9%
	Program	48 22%	20 9%	N/A
Corrective (4 subjects) (Total switches = 562)	Top-Down	N/A	67 12%	93 17%
	Situation	86 15%	N/A	112 20%
	Program	75 13%	129 23%	N/A
Enhancement (2 subjects) (Total switches = 299)	Top-Down	N/A	15 5%	77 26%
	Situation	28 10%	N/A	51 17%
	Program	64 21%	64 21%	N/A
General Understanding (3 subjects) (Total switches = 420)	Top-Down	N/A	30 7%	79 19%
	Situation	47 11%	N/A	92 22%
	Program	62 15%	110 26%	N/A

of code and understanding at the level of program and situation models is not efficient. Instead, generating a model top-down and switching to the program or situation models in instances where a more detailed understanding is necessary seems a more effective strategy than trying to build the mental models bottom up from the program. Another reason these subjects were able to switch often from the top-down model directly to the program model and back is related, we believe, to the fact that they were domain but not language experts. Domain expertise gives them the ability to make connections to the program code or situation model based on expectations from the domain.

### Corrective Maintenance

The corrective maintenance subjects spent the most time switching between program and situation models. These subjects were attempting to uncover defects in the code which requires that they understand the code at a low level, e.g. variable manipulation and data structure. What is even more interesting is that lack of domain expertise can intensify the effect: one subject, C1 was a domain novice and spent substantially more time

switching between situation and program models than the other three subjects who were domain experts. For **C1**, the debugging task and his lack of domain knowledge meant he spent over 50% of his switches going between program and situation model levels. Also, as expected, he spent slightly more time starting at the code level and abstracting to the situation model than he did starting at the situation model and switching back to code.

### **Enhancement**

Trends for the enhancement task are more difficult to detect. These two subjects switched frequently between the top-down and program models and between the program and situation models, but rarely between the top-down and situation models. Some of this may be more related to expertise since they were both language and domain experts. Having domain and language expertise allows one to develop connections directly to the program model based on expectations from the domain. Another interesting characteristic about both subjects is that they spent a substantial amount of time determining where in the code to add the enhancement and tracking down errors. Because of these particular tasks, their switching behavior is very similar to that of the corrective maintenance subjects.

### **General Understanding**

The group of General Understanding subjects consisted of three subjects. One of these subjects, **L1**, was a language only expert and switched the most between situation and program models. This seems to confirm the theory that novices must build up an understanding of the code in a bottom up manner. For the other two subjects, both language and domain experts, the story is different. One subject, **G2**, took a systematic approach to understanding, i.e. he read the code line by line, essentially building his mental representations bottom up. The other subject, **G1**, used an opportunistic strategy where he looked only at code that seemed interesting to him. **G2** had a fewer number of actual switches, tending to stay within a particular model for longer periods of time before switching to another model. (See Figure C.3 for action sequences over time.) **G1** had substantially more switches with a tendency to stay within a particular model a much shorter period of time (See Figure C.1). Both subjects shied away from switches from the

top-down to the situation model. This may be a reflection of the task in general. Whether the goal is to understand a piece of code or an entire system at a high level, if there is no goal beyond just understanding, then making connections between top-down and situation models may not be as important. Finding code that supports domain expectations may be enough, leaving the understanding of algorithms for a more specific task like debugging.

### B.3.1.2 Switches Between Models – Totals By Expertise

Table B.29 demonstrates model switching behavior for three groups of experts: language only, domain only, and language & domain.

Table B.29: Action Switches – Absolute & Percent of Total Switches by Expertise

<i>Expertise</i>	<i>Model</i>	<i>Model Switches – All Subjects</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
Language (2 subjects) (Total switches = 252)	Top-Down	N/A	23 9%	31 12%
	Situation	31 12%	N/A	68 27%
	Program	24 10%	75 30%	N/A
Domain (4 subjects) (Total switches = 533)	Top-Down	N/A	67 13%	94 18%
	Situation	83 15%	N/A	98 18%
	Program	78 15%	113 21%	N/A
Language & Domain (5 subjects) (Total switches = 717)	Top-Down	N/A	64 9%	172 24%
	Situation	90 13%	N/A	109 15%
	Program	147 20%	135 19%	N/A

#### Language Only Expertise

By far, the highest frequency of switches for language only experts is between program and situation models. This helps to confirm the hypothesis that when there is a lack of domain expertise, a mental model of the program is predominantly built from the bottom up.

#### Domain Only Expertise

Domain experts switch most often between situation and program models. Switches between situation and top-down models are also only slightly less frequent than those

between top-down and program models, reflecting the ability to make connections based on expectations. Overall, switches for domain experts are fairly evenly distributed across models. We surmise that, while they have the domain expertise, they do not have the language expertise and need to make connections to the program model or top-down model using the situation model as a bridge.

### **Language & Domain Expertise**

The highest rate of switching for language and domain experts is between top-down and program models. Again, this is not surprising since the domain expertise helps to make the connections to the code based on expectations. The other interesting result is that there are considerably fewer switches between top-down and situation models. We hypothesize that, unlike the domain only experts, language and domain experts do not necessarily need to use the situation model as a bridge between the top-down and program models. They have expectations based on domain knowledge and can directly make the connection into the program model without the help of an intermediate domain to translate the code. This is especially evident in the 9% rate of switches from the top-down model into the situation model.

#### **B.3.1.3 Switches Between Models – Totals By Accumulated Knowledge**

Table B.30 reports data on model switching behavior based on three levels of accumulated knowledge.

##### **Little Accumulated Knowledge**

Subjects with *little* accumulated knowledge switch most often into and out of the program model. It is difficult to determine the dominant influence for this group of subjects. Three of the subjects (C2, C3, and EN2) spent the majority of their time building a program model. They were trying to understand specific program behavior as it related to known defects. This required work at the code level. An alternate conjecture is that because they had little or no experience with the code, they could only build the representation at the program and situation model levels. We theorize that this is part of what the results represent, but we cannot conclude this from this set of subjects.



Table B.30: Action Switches – Absolute & Percent of Total Switches by Accumulated Knowledge

Accumulated Knowledge	Model	Model Switches – All Subjects		
		Top-Down Model	Situation Model	Program Model
Little (4 subjects) (Total switches = 701)	Top-Down	N/A	51 7%	161 23%
	Situation	83 12%	N/A	123 18%
	Program	128 18%	155 22%	N/A
Some (4 subjects) (Total switches = 488)	Top-Down	N/A	54 11%	69 14%
	Situation	67 14%	N/A	114 23%
	Program	58 12%	126 26%	N/A
Significant (3 subjects) (Total switches = 313)	Top-Down	N/A	49 16%	67 21%
	Situation	54 17%	N/A	38 12%
	Program	63 20%	42 14%	N/A

### Some Accumulated Knowledge

The majority of the switches that occurred for subjects with *some* experience with the code happened between program and situation models in both directions. This is probably due to task and expertise, rather than accumulated knowledge. Two subjects (**C1** and **C4**) were working on debugging tasks. One subject, **G2**, used a line-by-line systematic approach to understanding. Two subjects (**L1** and **C1**) were language only experts. Thus, all subjects had a reason to stay mostly within the program model regardless of the amount of prior experience.

Table B.30 shows that the distribution of switches among top-down and program and top-down and situation models is fairly even. What we expect to see in general for subjects with *some* accumulated knowledge is evenly distributed switches between all models. We conjecture that with *some* accumulated knowledge, programmers are able to make connections between all model levels based on what they already know. Further, because they have only partially built the mental representations, we would expect to see a higher frequency of switches among engineers with *some* accumulated knowledge than for those with *significant* accumulated knowledge. Those with *some* prior experience need to con-

tinue building their mental representations at all three levels as well as the connections between them. The average number of switches for those with *some* accumulated knowledge was 122 per subject. The average for those with *significant* accumulated knowledge was 104.

### **Significant Accumulated Knowledge**

The highest frequency of switching for subjects with *significant* accumulated knowledge occur between program and top-down models. Switching between top-down and situation models is lower and switching between program and situation models is the lowest. We conjecture that this behavior is related to amount of accumulated knowledge. For those with *significant* accumulated knowledge, models at all three levels will be close to complete and will need to be revised only due to changes to code or functionality. All three subjects in this group (AD1, AD2, and EN1) were focused on either adding functionality or porting code. Since they were very familiar with the code, they could easily update the program and top-down models to reflect changes that occurred as a result of the task. For example, knowing what functionality needs to be added (top-down model) is quickly followed by identifying a specific location to add it (program model).

### **B.4 Information Needs**

Information Needs are data necessary during a comprehension task. They can be determined directly from statements about needed information or indirectly through actions (such as looking for specific information) or when the programmer could obviously benefit if she had the data. The overall frequencies of each type of information needed for all subjects was reported in Chapter Five (see Tables 5.27, 5.28, and 5.29.). This section contains the details for those summarized results in section 5.5. Information needs are grouped according to task, expertise, and accumulated knowledge is reported.

Each table contains the top 30 information needs because the remaining (unreported) had only six or fewer references between all subjects. The information needs tables contain six or seven columns (depending on the number of subgroups: task has four, while expertise and accumulated knowledge have three each). The first contains the code used in analyzing the protocols for information needs. The second contains a brief description of the needed

information. The 3rd, 4th, 5th, and 6th (for task-related) contain the absolute number of references for each group of subjects. The last column is the total number of references (and the number used to calculate the percentage figures for each cell in the row) to the information need.

#### B.4.1 Information Needs – By Task

Tables B.31 and B.32 contain the information needs data by task. We see that overall, *I7* (Domain concepts) is most frequent. However, it is only the most frequent for two of the four tasks, corrective and adaptive maintenance. For General Understanding, it is the second most important type, and for Enhancement it is the fifth. The reasons for this is not readily apparent. Accumulated knowledge may be a factor. Of the five total references to *I7*, four are attributable to **EN2**. While **EN2** was a domain and language expert, he had *little* accumulated knowledge about the program. Therefore, he may have required the additional domain information to help build up his top-down model of the program.

Nothing can be concluded about the type of information required during adaptive maintenance except that these two subjects had the fewest number of information needs than any of the other tasks (the average number of information needs was 34 each, during the programming session). We hypothesize that this is more a function of both expertise and accumulated knowledge. Both were domain experts and both had significant knowledge about the program. The combination of expertise and experience would seem to negate the need for a lot of additional information either on the system itself or the domain.

For corrective maintenance, task related information is more apparent. The first two most frequent information needs, *I7* (Domain concepts) and *I61* (model connected information) are important in understanding bugs (as they are in other tasks). What seems more indicative of the task, however, is that *I4* (location and uses of identifiers), *I9* (list of browsed locations), *I2* (function called by), *I4* (call graph display), and *I73* (bug behavior isolated) are the next most frequent information needs. Discovering information about the program at the code level is crucial to understanding a defect.

The enhancement tasks' most important information need was *I9* (recently browsed locations). Both subjects needed to get back to some place they had recently been and

wanted to know the location more often than they needed other information. It is difficult to draw conclusions for these subjects because, overall, they had a low number of information needs (average per subject over programming session = 48). Compare this to debugging (average per subject = 73.5) and General Understanding (average per subject = 81.67). Both enhancement task subjects were language and domain experts and the need for domain concept information was this group's fifth most important information need. The other interesting result for these subjects is that the second and third most frequent information needs are similar to the debugging task (*I61* and *I4*). Recall that earlier we said that the enhancement subject spent a significant amount of time tracking bugs they introduced during the enhancement task. Therefore, this behavior makes sense for this group.

The top information needs for General Understanding are not surprising. The top two are domain knowledge (*I7*) and connected model information (*I61*). When there is no specific goal other than to understand the program (perhaps for future maintenance tasks), this type of information helps to build mental representations at the domain level as well as at other levels depending on the individual preferences and strategies of the engineer. What is interesting is that connected model information (*I61*) was not as important to **G2** who was the person who read the code line by line during the programming session. He had only three references to *I61*. On the other hand, he had about the same number of references to *I7* (Domain concepts) as the other two subjects, which indicates **G2** was interested in understanding domain concepts. The third most frequent information need was *I3* (Highlighted begin/ends of control blocks), which was mainly a result of **G2**'s line by line strategy. He was responsible for 11 of the 14 references. The next two information needs, *I5* (data structure information) and *I4* (Location and uses of identifiers) is also typical information that is required, at least for the major components of a program. The 6th most frequent type of information need for this group was *I68* which concerns design decisions. Determining what decisions were made during design can help one to understand a great deal about the program, including implementation tradeoffs and anomalies. Design decisions are high-level information, so it is not surprising that the majority of the references were made by **G1**, the engineer who looked at the code opportunistically to "get a feel for what it does".

Table B.31: Information Needs Frequency By Task

<i>Code</i>	<i>Information Need</i>	<i>Adaptive Task</i>	<i>Corrective Task</i>	<i>Enhance Task</i>	<i>Understand Task</i>	<i>Ttl</i>
I7	Domain concept descriptions	11	41	5	28	85
I61	Connected domain-program-situation model knowledge	0	33	7	30	70
I4	Location and Uses of identifiers	1	26	9	11	47
I9	List of browsed locations	1	17	11	5	34
I5	Format of data structure plus descriptions	2	8	0	13	23
I43	General classification of routines/functs	1	11	6	5	23
I2	Function called By	1	14	3	3	21
I1	Variable definitions/declarations	1	2	1	13	17
I3	Highlighted begin/ends of control blocks	0	2	1	14	17
I22	History of past modifications	6	7	2	2	17
I73	Bug behavior isolated	1	11	5	0	17
I14	Call Graph Display	0	12	1	3	16
I68	Design Issues/decisions	0	3	4	9	16
I24	Executed statements, procedure calls, var values	4	7	3	0	14
I42	Utility functs definitions	0	0	1	13	14
I27	Directory layout organization	5	5	0	2	12
I17	Location of desired code	4	1	5	0	10
I6	Location/description lib/sys calls	1	3	0	5	9
I32	Common objects not used in traditional way	2	1	1	5	9
I58	High-level description of how code is laid out	0	4	0	5	9
I67	Location to put changes	0	0	9	0	9
I20	Documentation list & location	0	3	0	5	8
I44	Routines that do most domain work	0	5	0	3	8

Table B.32: Information Needs Frequency By Task (continued)

<i>Code</i>	<i>Information Need</i>	<i>Adaptive Task</i>	<i>Corrective Task</i>	<i>Enhance Task</i>	<i>Understand Task</i>	<i>Ttl</i>
I72	Good direction, possible program segments	0	6	0	2	8
I16	Naming conventions for system & libs	0	4	0	3	7
I36	Location sequence where ID is used	0	5	1	1	7
I53	IFDEF'd behavior, conditions which branch is taken	3	0	0	4	7
I66	Expected program state, var values when funct called	0	6	0	1	7
I74	Bug description, why/how other fixes were done	2	3	0	2	7
I79	Ripple Effect of change	6	0	1	0	7

#### B.4.2 Information Needs – By Expertise

Tables B.33 and B.34 contain information needs grouped by expertise.

Domain concepts (*I7*) and connected model information (*I61*) are top information needs for language only experts. With language only expertise there is a need for domain knowledge. Because these subjects understand the language, it makes sense that making connections from the program model to the other models is important. The more interesting results for this group, however, is that the next few most important information needs are all focused towards program model information. These are *I5* (data structures), *I2* (function called by), *I4* (location and uses of identifiers), and *I3* (highlighted begins and ends of control blocks). This is probably due to their lack of domain expertise. Both subjects, **C1** and **L1**, spent the majority of their time in the program model, at the level in which they were comfortable. Assuming that they mainly concentrate on building a program model, they would require information at that level. Thus, the hypothesis is that language experts need to understand domain concepts and the connections between models, but they are most comfortable working within the program model (at least until they become domain experts) and will need a lot of low level information directly concerned with the code.

It is difficult to draw conclusions about the domain experts since three of the four subjects (**C2**, **C3**, and **C4**) were working on a debugging task. The influence of task is apparent not only in the high frequency of code level information needs (*I4* and *I9*) but also because the fifth most important information need is *I73*, bug behavior isolated.

Language and domain experts need a wider variety of information at all levels. Not only is domain concept information and connections between models important for this group, but also location and uses of identifiers (*I4*), a list of browsed locations (*I9*), utility functions identified and defined (*I42*), variable definitions and declarations (*I1*), and a general classification of routines and functions (*I43*).

Table B.33: Information Needs Frequency By Expertise

<i>Code</i>	<i>Information Need</i>	<i>Language Only</i>	<i>Domain Only</i>	<i>Language &amp; Domain</i>	<i>Ttl</i>
I7	Domain concept descriptions	23	27	35	85
I61	Connected domain-program-situation model knowledge	17	28	25	70
I4	Location and Uses of identifiers	7	23	17	47
I9	List of browsed locations	2	15	17	34
I5	Format of data structure plus descriptions	9	6	8	23
I43	General classification of routines/functs	4	8	11	23
I2	Function called By	8	8	5	21
I1	Variable definitions/declarations	2	3	12	17
I3	Highlighted begin/ends of control blocks	5	0	12	17
I22	History of past modifications	4	9	4	17
I73	Bug behavior isolated	1	10	6	17
I14	Call Graph Display	3	9	4	16
I68	Design Issues/decisions	3	2	11	16
I24	Executed statements, procedure calls, var values	0	9	5	14
I42	Utility functs definitions	0	0	14	14
I27	Directory layout organization	3	5	4	12
I17	Location of desired code	0	2	8	10
I6	Location/description lib/sys calls	0	3	6	9
I32	Common objects not used in traditional way	0	3	6	9
I58	High-level description of how code is laid out	1	3	5	9
I67	Location to put changes	0	0	9	9
I20	Documentation list & location	4	3	1	8
I44	Routines that do most domain work	1	5	2	8



Table B.34: Information Needs Frequency By Expertise (continued)

Code	Information Need	Language Only	Domain Only	Language & Domain	TU
I72	Good direction, possible program segments	1	5	2	8
I16	Naming conventions for system & libs	0	4	3	7
I36	Location sequence where ID is used	0	5	2	7
I53	IFDEF'd behavior, conditions which branch is taken	0	3	4	7
I66	Expected program state, var values when funct called	2	4	1	7
I74	Bug description, why/how other fixes were done	3	2	2	7
I79	Ripple Effect of change	0	6	1	7

### B.4.3 Information Needs – By Accumulated Knowledge

Tables B.35 and B.36 contain information needs grouped by amount of accumulated knowledge.

Subjects with *little* accumulated knowledge need domain information (I61) to help make connections (I7) to other models. This is expected for this group because they have spent little time with the code and do not yet have the mental models required to do their tasks efficiently. The next most frequent information needs are *I9* (list of browsed locations), *I4* (location and uses of identifiers), *I43* (general classification of routines and functions), *I42* (utility functions identified and defined), and *I14* (Call Graph). Most of these are appropriate for gaining an understanding of code when little is known about it. All five of these information needs were needed more often by this group than the others. For instance, the average number of *I4* (call graph) for those with *little* accumulated knowledge was 3.25, for those with *some* it was .75, and those with *significant* accumulated knowledge had no need for this information need. Thus, engineers with *little* accumulated knowledge about the code tend to need types of information those with more experience no longer need. This group also had a higher total average number of information needs in general. Each subject had an average of 80 information needs during the programming

session. Those with *some* experience had an average of 67.5 while those with *significant* accumulated knowledge had an average of 37.67.

Conclusions about information needs for people with *some* accumulated knowledge about the program are difficult to draw from our sample. Three of the subjects (**C1**, **L1**, and **G2**) worked mostly at the program model level, and this is reflected in their information needs. Aside from needing domain concept and model connection information, these subjects required information on location and uses of identifiers (*I4*), highlighted begins and ends (*I3*), data structure information (*II5*), and variable definitions/declarations (*I1*). One could argue that this is expected with *some* experience since they still have much to learn about the program. In this regard, they are closer to those with *little* accumulated knowledge than those with *significant* experience. While the group with *some* experience had some of the same needs as those with *little* accumulated knowledge, their information requirements are fewer. The hypothesis for the programmers with *some* accumulated knowledge is that information needs are somewhere between those with *little* and those with *significant* accumulated knowledge. This means less one-time needs like call graph or general classifications of functions than the subjects with *little* accumulated knowledge and more information about model connections than the subjects with *significant* accumulated knowledge.

Overall, subjects with *significant* accumulated knowledge do not need as much information as the other two groups. *I61* (connected model information) was the 6th most important information need for this group. This implies that the majority of the connections are already there. Instead, this group's information needs seem tied more directly to their specific task. For example, *I4* (location and uses of identifiers), *I73* (bug behavior isolated), *I22* (history of past modifications), and *I17* (location of desired code) are related to **AD1**, **AD2**, or **EN1**'s adaptive or enhancement task.

Table B.35: Information Needs Frequency By Accumulated Knowledge

<i>Code</i>	<i>Information Need</i>	<i>Little Knowledge</i>	<i>Some Knowledge</i>	<i>Significant &amp; Knowledge</i>	<i>Ttl</i>
I7	Domain concept descriptions	34	39	12	85
I61	Connected domain-program-situation model knowledge	40	26	4	70
I4	Location and Uses of identifiers	20	19	8	47
I9	List of browsed locations	24	3	7	34
I5	Format of data structure plus descriptions	8	13	2	23
I43	General classification of routines/funcs	17	5	1	23
I2	Function called By	12	8	1	21
I1	Variable definitions/declarations	4	12	1	17
I3	Highlighted begin/ends of control blocks	0	16	1	17
I22	History of past modifications	3	7	7	17
I73	Bug behavior isolated	5	6	6	17
I14	Call Graph Display	13	3	0	16
I68	Design Issues/decisions	11	5	0	16
I24	Executed statements, procedure calls, var values	1	7	6	14
I42	Utility functs definitions	14	0	0	14
I27	Directory layout organization	1	6	5	12
I17	Location of desired code	4	0	6	10
I6	Location/description lib/sys calls	3	5	1	9
I32	Common objects not used in traditional way	4	3	2	9
I58	High-level description of how code is laid out	8	1	0	9
I67	Location to put changes	4	0	5	9
I20	Documentation list & location	4	4	0	8
I44	Routines that do most domain work	7	1	0	8

Table B.36: Information Needs Frequency By Accumulated Knowledge (continued)

<i>Code</i>	<i>Information Need</i>	<i>Little Knowledge</i>	<i>Some Knowledge</i>	<i>Significant &amp; Domain Knowledge</i>	<i>Ttl</i>
I72	Good direction, possible program segments	7	1	0	8
I16	Naming conventions for system & libs	6	1	0	7
I36	Location sequence where ID is used	6	0	1	7
I53	IFDEF'd behavior, conditions which branch is taken	4	0	3	7
I66	Expected program state, var values when funct called	0	7	0	7
I74	Bug description, why/how other fixes were done	0	5	2	7
I79	Ripple Effect of change	1	0	6	7

## Appendix C

### PROGRAMMING SESSIONS – INDIVIDUAL SUBJECT RESULTS

This chapter contains the results of analysis of several individual protocols. Each section contains a brief biography of the maintenance engineer and the task performed. Tables summarizing the results for each subject include action counts, switches, hypothesis types, and information needs. All tables are organized in the same manner as the corresponding tables found in Chapter Five or Appendix B. These tables are the groundwork for the rest of this study.

The *Action-Type Frequency* tables contain lists of the identified action types for each model. Frequencies are organized by a *tag* used for encoding protocols (see Appendix A). These tables contain the total number of references to each action and their frequency by model level. The tables also include the total number of actions for each model level. Actions with zero frequencies do not appear in the table.

*References and Action Switches between Models* tables demonstrate switching activity generated by actions and indicate which model the programmer was actively constructing during the programming session. The rows in each table indicate the starting model and the column represents the ending model. Each table contains the total number of references to each model component to highlight the difference between the total number of actions in each model and those that actually caused a switch.

Action Sequence graphs illustrate model switching behavior. Within each graph, model levels are represented on the Y-axis and time is represented on the X-axis. Time steps represent action counts over time. For example, Figure C.1 demonstrates that subject **G1** started the programming session in the Top-Down model. The next 18 or so actions occurred in the Top-Down model. The 20th action was a Situation model action. This is indicated by the jump in the graph. Looking at these graphs gives us a method

for visualizing switching behavior. A similar graph is provided for hypothesis generated switches.

Subject data also includes tables for hypothesis types. The *Tags* column contains the codes used during protocol analysis. The *Hypothesis Type* column briefly describes the hypothesis type and the last column give both the number of references to each hypothesis type and the percent of total hypotheses in the associated model. The tables also contain rows for total hypotheses per model. Hypothesis types with zero frequencies do not appear in the table. Similar to the action generated switching tables, hypothesis generated switching tables are included in the hypotheses section for each subject.

Information Needs tables are provided on an individual subject basis. The column labeled *Code* contains the code used during protocol analysis for identifying information needs. The *Information Need* column provides a description of the information need. The third column, *Action Codes*, lists model actions related to each need. The last column show how often this information was required during the programming session. Information needs with zero frequencies do not appear in the table.

Prior to each programming session, subjects were asked to find a task they knew they would eventually have to complete in their normal jobs. Each session lasted two hours during which subjects talked aloud while attempting to perform the task. We recorded the session and later transcribed it for analysis. Each engineer's statements were studied for a verification of the integrated model, action identification and counts, action and hypothesis generated switches between models, hypothesis type identification and counts, and information needs.

### **C.1 General Understanding**

Two protocols for general understanding were analyzed, **G1** and **G2**. General understanding is defined as learning the system for the purpose of understanding. Perhaps the engineer has just acquired maintenance responsibility. The objective of the first engineer was to understand an entire system and he applied an opportunistic approach to this task. Understanding a single module using a systematic approach was the focus of the second engineer.

### C.1.1 G1: Domain/Language Expert – General Understanding

G1 had chosen a piece of code of approximately 40,000 lines. He was also in the process of learning a new CASE tool called PROCASE. This tool had several features which were extremely helpful to the engineer during the comprehension session. Examples of features are call-graph display of functions, text re-formatting, limited cross-referencing capability for viewing variable declarations and use, and pruning & source text elision. The engineer started the session with three major knowledge components. He had high-level knowledge of program function as well as knowledge of the enhancement he was supposed to provide. He had not looked at any documents other than the external requirements specification, which is where he obtained the high-level knowledge of the program's functionality. He had also looked at the directory containing the code so he knew the names of the files for which he would be responsible.

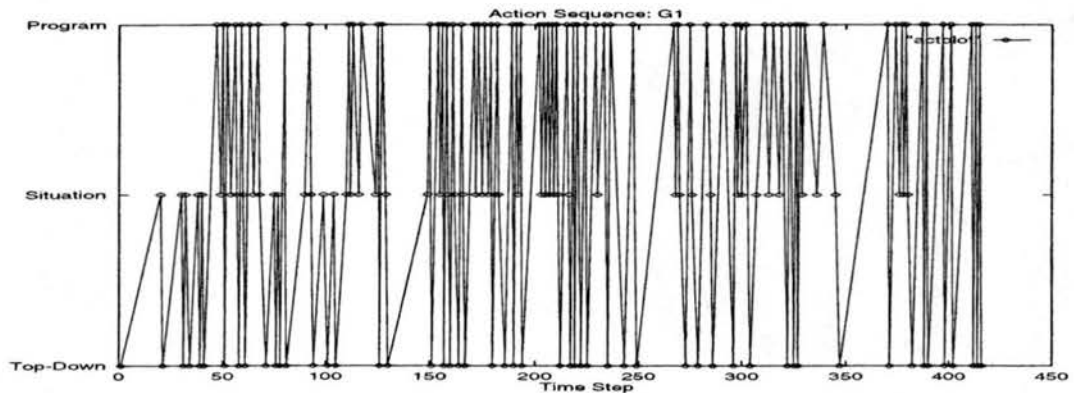


Figure C.1: G1: General Understanding – Action Sequence

Table C.1: G1: References and Action Switches Between Models

Number of References	Model	Model Switches – Understand Program		
		Top-Down Model	Situation Model	Program Model
231	Top-Down Model	N/A	14	38
79	Situation Model	25	N/A	30
114	Program Model	27	41	N/A

Table C.2: G1: Action-Type Frequencies – General Understanding

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	6	3%
OP2	Determine next prgm. seg to examine	21	9%
OP3	Generate/revise hypothesis re: functionality	62	27%
OP4	Determine relevance of prgm segment	18	8%
OP6	Determine understand strategy	22	10%
OP7	Investigate oversight	1	0%
OP8	Failed hypothesis	3	1%
OP13	Study/initiate program execution	3	1%
OP15	Generate questions	11	5%
OP16	Answer questions	2	0%
OP17	Chunk & store knowledge	18	8%
OP20	Generate task	30	13%
OPCONF	Confirmed hypothesis	6	3%
OPKNOW	Top-down knowledge	28	12%
<i>Total</i>	<i>Top-Down Model Actions</i>	231	54%
SIT2	Develop questions	6	8%
SIT4	Chunk & store	25	32%
SIT5	Determine relevance of sit. know.	1	1%
SIT7	Generate hypothesis	30	38%
SIT8	Determine understand strategy	1	1%
SIT10	Failed hypothesis	1	1%
SITKNOW	Situation model knowledge	15	19%
<i>Total</i>	<i>Situation Model Actions</i>	79	19%
SYS1	Read intro code comments/related docs	27	23%
SYS2	Determine next prg segmt to examine	3	3%
SYS3	Examine next module in sequence	33	29%
SYS5	Examine data structs & definitions	3	3%
SYS7	Chunk & store knowledge	13	11%
SYS8	Generate hypothesis	7	6%
SYS10	Determine understand strategy	7	6%
SYS11	Generate new task	1	1%
SYS12	Generate question	3	3%
SYS21	Mental simulation	1	1%
SYSCONF	Confirmed hypothesis	2	2%
SYSKNOW	Program model knowledge	14	12%
<i>Total</i>	<i>Program Down Model Actions</i>	114	27%

Table C.3: G1: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
54	Top-Down Model	N/A	14	2
30	Situation Model	13	N/A	4
7	Program Model	3	3	N/A



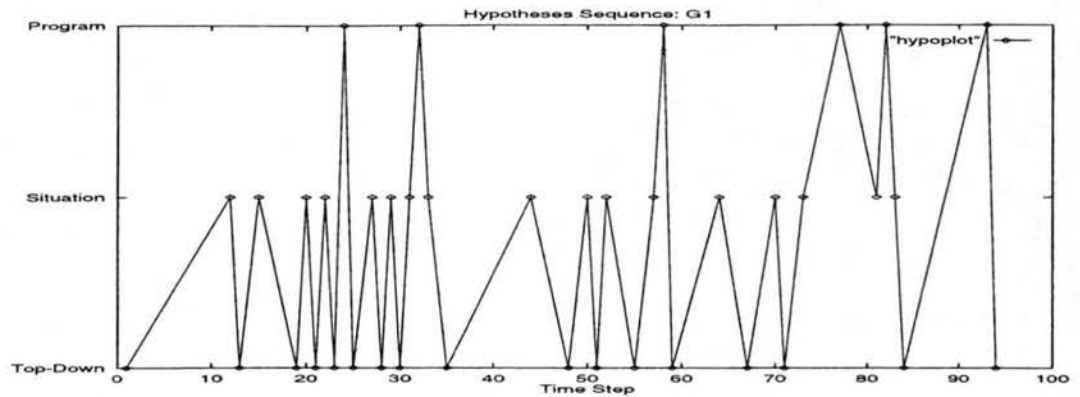


Figure C.2: G1: Hypotheses Sequence

Table C.4: G1: Hypothesis-Type Frequencies – General Understanding

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	20	37%
	OPH2	Variable functionality/domain concepts	3	5%
	OPH3	Rules of discourse/Expectations	1	2%
	OPH5	Existence of installed (running) program	1	2%
	OPH13	Number/type/location of file	2	4%
	OPH14	Available functionality	10	18%
	OPH16	Level & structure of code/scope	15	28%
	OPH17	Design Decisions/Modifications	2	4%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	54	50%
Situation Model	SITH1	Variable function	7	23%
	SITH2	Function/code block execution order/state	2	7%
	SITH3	Function/procedure function, call function	11	37%
	SITH7	Existence of functionality/algorithm/variable	8	26%
	SITH8	Program function	2	7%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	30	33%
Program Model	SYSH1	Variable function	3	44%
	SYSH4	Variable structure	1	14%
	SYSH7	Variable value/defaults	1	14%
	SYSH12	Variable definition & it's location	1	14%
	SYSH14	Code block function	1	14%
	<i>Total</i>	<i>Program Model Hypotheses</i>	7	8%

Table C.5: G1: General Understanding - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	10
I61	Connected domain-program-situation model knowledge	SYS7	15
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	2
I9	List of browsed locations	OP2, OP4, SYS2 OP4	4
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	4
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	4
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	2
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	3
I14	Call Graph Display	OP1, OP2, SYS3	3
I68	List of issues/decisions considered during design	SYS12	6
I42	Utility functions definitions and comments explaining why it was rewritten	OP2	13
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	2
I58	High-level description of how code is laid out	OP3	5
I20	Documentation list and location	OP1	1
I44	List of routines that do most of the domain-type work	OP3	2
I72	Good direction to follow given what is already known, possible program segments to examine	SYS10	2
I16	Naming conventions separated by system or library objects that use them Rules used for new naming new procedures	OP3, SYS3, SYS8 SIT7	2
I36	Sequence of locations where ID is used	OP2	1
I53	IFDEF'd behavior, conditions under which the branch is taken or not	SIT2	4
	Language enhancements		
I12	Environment, global, local scope	SYS3	2
I29	Naming conventions for program	SYS1, SYS3	1
I39	Main program location	OP1	2
I41	Call graph with extraneous information not relevant elided	OP2	2
I48	Code formatted in expected way	SYS3	3
I19	Acronym definitions	SIT1, SIT7	1
I45	List of internal support routines	OP3	2
I47	List of identifiers and domain concepts that are important	OP4	2
I49	File name of current file	SYS3	1
I55	Domain Simulation	SIT2	1
I10	Function call count	OP1, SYS2	1

Table C.6: G1: General Understanding – Information Needs (continued)

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I26	All include file definitions and uses	OP1, SYS5	1
I28	List of all routines with initialization code	OP2, SYS2	1
I46	What kind of architecture dependent code currently exists and is functional what architecture hooks exist so code can be easily added.	OP3	1
I51	What happens to read in data	SYS5	1

### C.1.2 G2: Domain/Language Expert – Understand One Module

G2 was in the process of understanding one module in a system for which he recently took over responsibility. The entire program consists of approximately 85 or 90 KLOC (thousand lines of code). During the programming session he was interested in thoroughly examining the main procedure which controlled a majority of the system. The system was a terminal emulation program that uses a X.25 networking protocol for communications. The program is written in a non-standard Pascal. The engineer was a software networking expert (six years professional experience), however, he had only been working with the X.25 protocol for less than one year. He was very familiar with three different versions of Pascal as well as MS-DOS assembly language.

G2 preferred to work with a hard copy of the code and resorted to using a computer for tasks like searching for variable use/definitions and for writing mail messages to other experts when he was unable to answer certain questions himself. He annotated the hard copy with important information which he frequently referred to during the session. He also took a very systematic approach to learning. He followed the code, line by line and attempted to understand everything before going on to the next line. At the end of the two hours he had successfully understood the entire module with very few unanswered questions.

Table C.7: G2: Action-Type Frequencies – General Understanding

<i>Tag</i>	<i>Action-Type</i>	<i>Th. Refs</i>	<i>% Th.</i>
OP1	Gain high-level Program overview	23	39%
OP2	Determine next prgm. seg to examine	1	2%
OP3	Generate/revise hypothesis re: functionality	13	22%
OP6	Determine understand strategy	3	5%
OP15	Generate questions	4	6%
OP17	Chunk & store knowledge	1	2%
OP20	Generate task	6	10%
OPKNOW	Top-down knowledge	8	14%
<i>Total</i>	<i>Top-Down Model Actions</i>	59	17%
SIT1	Gain situation knowledge	11	33%
SIT4	Chunk & store	8	25%
SIT7	Generate hypothesis	7	21%
SITKNOW	Situation model knowledge	7	21%
<i>Total</i>	<i>Situation Model Actions</i>	33	10%
SYS1	Read intro code comments/related docs	18	7%
SYS2	Determine next prg segmt to examine	6	2%
SYS3	Examine next module in sequence	57	23%
SYS4	Examine next module in cntrl-flow	1	1%
SYS5	Examine data structs & definitions	3	1%
SYS7	Chunk & store knowledge	35	14%
SYS8	Generate hypothesis	40	16%
SYS10	Determine understand strategy	9	4%
SYS11	Generate new task	15	6%
SYS12	Generate question	11	5%
SYS13	Determine if looking at right code	1	1%
SYS19	Failed hypothesis	2	1%
SYS21	Mental simulation	5	2%
SYS23	Search for var defines/use	4	1%
SYS24	Search for block begin/end	10	4%
SYSCONF	Confirmed hypothesis	2	1%
SYSKNOW	Program model knowledge	28	11%
<i>Total</i>	<i>Program Down Model Actions</i>	247	73%

Table C.8: G2: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
59	Top-Down Model	N/A	7	24
33	Situation Model	9	N/A	23
247	Program Model	22	26	N/A

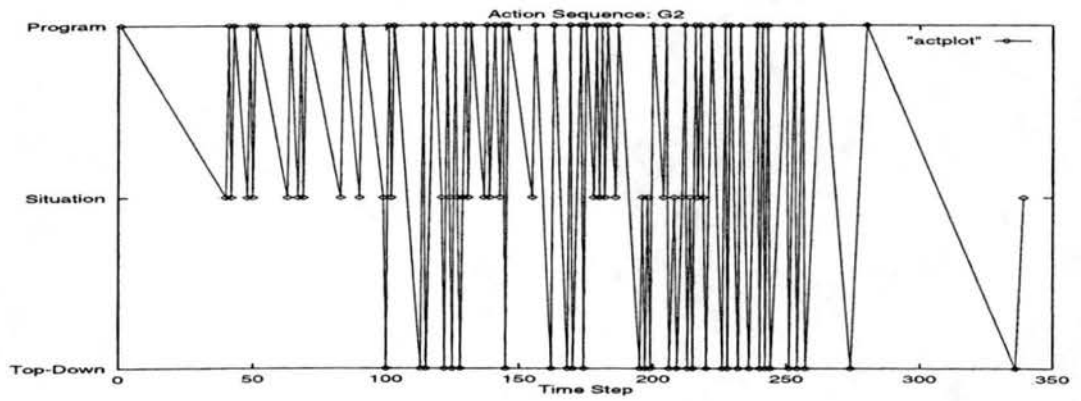


Figure C.3: G2: Understand One Module – Action Sequence

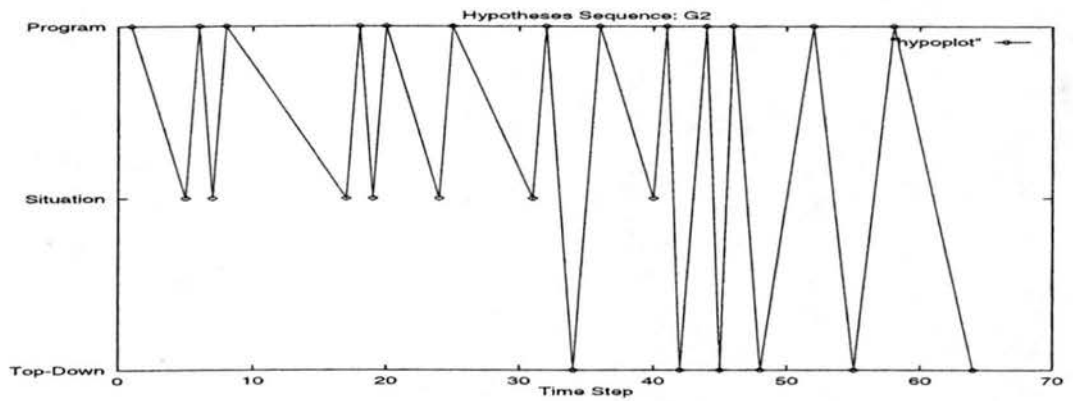


Figure C.4: G2: Understand One Module – Hypotheses Sequence

Table C.9: G2: Hypothesis-Type Frequencies – Understand One Module

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	7	54%
	OPH2	Variable functionality/domain concepts	2	15%
	OPH3	Rules of discourse/Expectations	3	23%
	OPH4	I/O behavior	1	8%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	13	22%
Situation Model	SITH1	Variable function	4	58%
	SITH2	Function/code block execution order/state	1	14%
	SITH3	Function/procedure function, call function	1	14%
	SITH7	Existence of functionality/algorithm/variable	1	14%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	7	11%
Program Model	SYSH1	Variable function	7	17%
	SYSH2	Function/procedure function	4	10%
	SYSH3	Procedure/function call behavior	1	3%
	SYSH4	Variable structure	6	15%
	SYSH5	Location/type/existence of function call	1	3%
	SYSH6	Statement execution order/state	5	12%
	SYSH7	Variable value/defaults	3	7%
	SYSH9	Variable/construct equivalency	1	3%
	SYSH10	Syntax meaning	1	3%
	SYSH11	Design decisions	2	5%
	SYSH12	Variable definition & it's location	1	3%
	SYSH14	Code block function	2	5%
	SYSH19	Code block boundary location	4	10%
SYSH20	Params/type definitions in procedure call	2	4%	
	<i>Total</i>	<i>Program Model Hypotheses</i>	40	67%

Table C.10: G2: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches - Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
13	Top-Down Model	N/A	0	5
7	Situation Model	0	N/A	7
40	Program Model	6	7	N/A

Table C.11: G2: Understand One Module - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	9
I61	Connected domain-program-situation model knowledge	SYS7	3
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	5
I9	List of browsed locations	OP2, OP4, SYS2 OP4	1
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	4
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	1
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	8
I3	Highlighted begin/ends of control blocks	SYS3, SYS7, SYS8, SYS10, SYS24	11
I22	History of past modifications	SYS3	1
I68	List of issues/decisions considered during design	SYS12	1
I6	Location and description of library routines and system calls	OP1, OP2, OP20, SYS11	5
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	3
I16	Naming conventions separated by system or library objects that use them Rules used for new naming new procedures	OP3, SYS3, SYS8 SIT7	1
I66	Expected program state, e.g. expected variable values when procedure is called	SYS11	1
I11	Language conventions that are different from standard conventions Language enhancements	SYS1, SYS3, SYS10	4
I18	I/O parameters, definitions, examples of calls with explanation, what it is used for	SYS3, SYS5, SYS8	1
I29	Naming conventions for program	SYS1, SYS3	4
I30	Where variable is toggled, when and why, where passed to and why	SYS2, SYS12	1
I48	Code formatted in expected way	SYS3	1
I45	List of internal support routines	OP3	2
I47	List of identifiers and domain concepts that are important	OP4	1
I71	Concise error description	SYS20	1
I10	Function call count	OP1, SYS2	1
I26	All include file definitions and uses	OP1, SYS5	1
I28	List of all routines with initialization code	OP2, SYS2	1
I52	User defined code block identification	SYS7	3
I63	All definitions and uses of a variable prioritized..so more important uses show up first	SYS10	1
I54	List of experts	SYS10	2
I59	How a variable is passed into a procedure, e.g. by value, address, etc.	SYS1	1
I35	Any difference between variable name and use	SYS8	1
I56	The include file a particular structure is defined in	OP2	1

## G2: Understand One Module – Processes

This section contains the results of analyzing the protocols for comprehension processes for G2. Processes are identified at the episode, aggregate-level, and session levels. Each process level can be thought of as a higher level abstraction of the previous level. These are presented in graph form as state machines. Lists of the sequences of states are provided to illustrate paths through the graph. These are the actual sequences found during the analysis of protocols.

Tables for process trigger frequencies are included and provide an indication of the importance and identification of connections between processes.

Also provided are the information needs and frequencies found for each process at the episode level and the number of times a particular process occurred in the protocol.

### Episode Level Processes – Understand One Module

Episodes embody a sequence of activities carried out to accomplish a goal. Episodes containing common elements with similar goals emerge as a process. To illustrate, in our example protocol a particular episode classified as *P1* starts with the overall goal of understanding a specific block of code, e.g. “I’m going to read the description and see if it gives me some good clues as to what’s going on.” Some of the observed activities that support the original goal are: generating hypotheses while reading comments, chunking information, and making note of interesting aspects and postponing investigation of them.

Table C.12 lists seven episode-level processes and how often they occurred in one example transcript (subject G2 (see table 4.1 )) The table shows that our subject spent the majority of his time reading the code, determining the behavior of a variable, and

Table C.12: G2: Process Frequency Count

<i>Process Name</i>	<i>Process Code</i>	<i>Frequency</i>
Read Block in Sequence	P1	13
Integrate Partially Understood	P2	5
Determine Variable Def/Use	P3	7
Incorporate Acquired Program Knowledge	P4	5
Identify Block Boundaries	P5	2
Resolve Deferred Questions	P6	2
Understand a Procedure Call	P7	8



incorporating this knowledge into his mental model of the program module. The engineer applied a systematic strategy of reading each line of code in approximate sequence. Thus, references to processes P1, P3, and P4 support building the program model. We also included these processes as state machine diagrams to illustrate the basic form of the episodes.

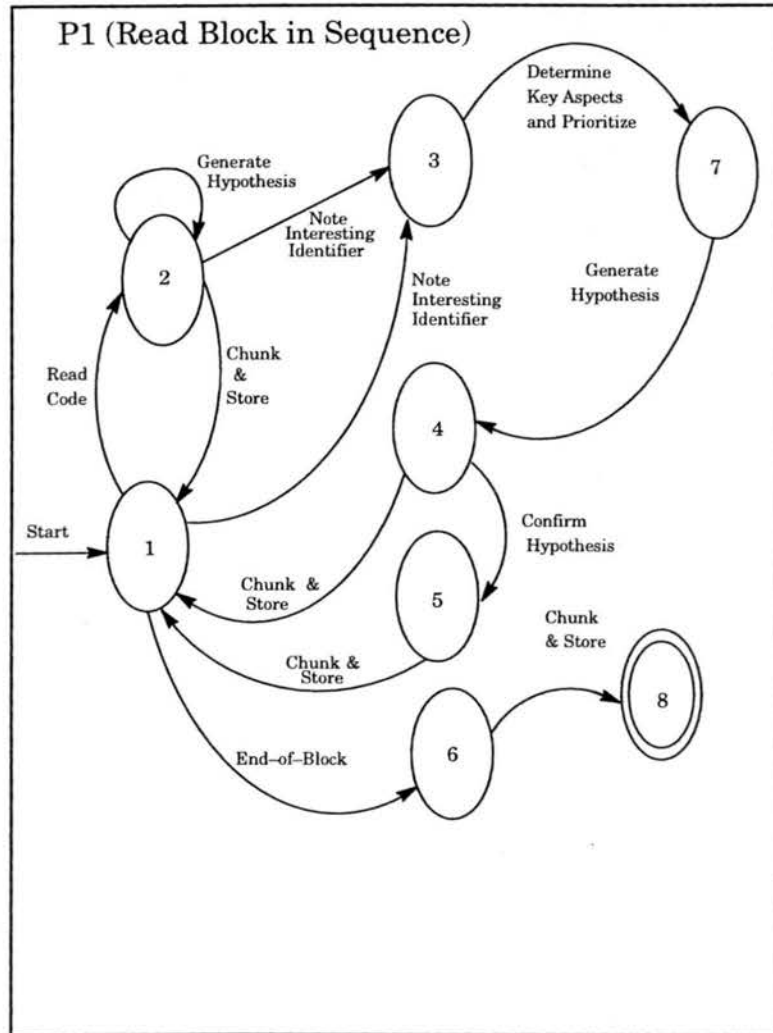


Figure C.5: G2: Process 1 – Read Block

The *Read Block in Sequence* process models a progression of program model actions that include reading a logical chunk of code and making note of important identifiers. Relevance can later be determined from a list of key aspects organized according to some priority scheme. The following excerpt taken from the protocol illustrates P1.

**SYS4**

So when the routine starts, what does it do? It says look and see if the function, the call, the user input line which is passed into this is an "F2" or "F3" function. So it says, IF the first thing that it found was "F2" or IF command line match of the token with the literal F2 bracket bracket, or if the command line match was CANCEL then close the file; but remember the name

**SYS11**

because I'm going to want to know what CLOSE-FILE actually does.

**SYS8**

It looks like it's probably an internally called routine.

**SYS11**

So I'll go look for that in a second.

**SYS3**

Otherwise use VFMT to um, write a null, looks like a null to the STREAM.\$STDOUT and that is the end of that IF statement.

**SYS7**

So that's what it does if it's "F2".

Not all the activities represented by arcs in the diagram are found in every process identified as P1. For example, the above citation is actually a part of a larger P1 episode. In the larger episode, instead of reaching the *End-of-Block* and *Chunking & Storing* followed by the stopping state, the path takes the process to the starting state and begins again.

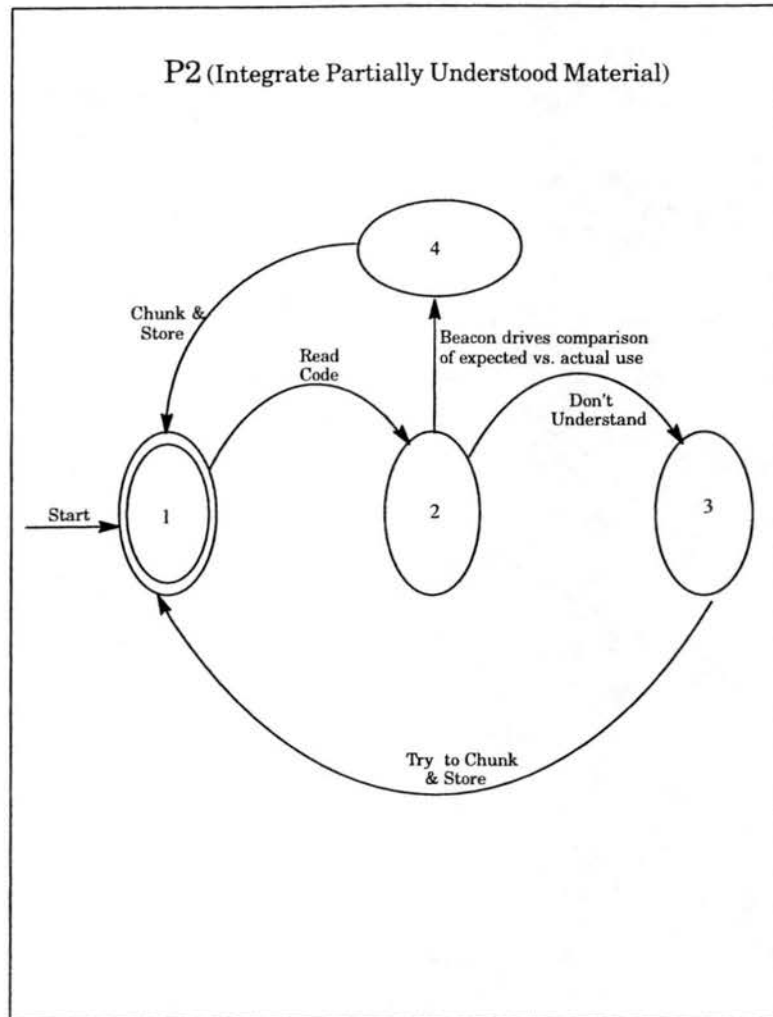


Figure C.6: G2: Process 2 – Integrate Partially Understood

The *Integrate Partially Understood* process attempts to integrate information that is not entirely understood by the programmer. Rules of discourse and beacons can drive this process when variable or procedure names do not meet expectations. An example taken from the protocol illustrates P2:

**SYS2**

So, window, there's my other window. One, (click click of the keyboard). We're looking for CLOSE-FILE.

**SYS1, SYS3**

CLOSE-FILE is an internally defined routine and it says close the currently opened file if any, there is not an open

file, then we're just going to return. If there is an open file that's TRANSMIT or PASTE then we'll close the file, set some variables to zero. If the open file was XMIT then we're going write XMIT file whatever it's name was, closed. Set, reset the number of events, okay,

**SYS12**

I'm not sure what that means.

**SYS3**

Um, otherwise it's not one of those two kinds of files then we're going to close. This other file stream file um set the access mode to an appropriate thing and write out "RECEIVED FILE IS CLOSED" and END. So, and then set open file to NONE.

**SYS7** So all that routine does is close some sort of a special stream file and it looks like there are three kinds of stream files. And it just closes whatever one that is. So that tells me a little bit about that.

Integrating partially understood information happens at all levels of abstraction. The above episode is an example of an occurrence at the program model level. Another was found at the top-down model level and one of the other occurrences was at a very low program model level evident by the dialogue concerning loop variables.

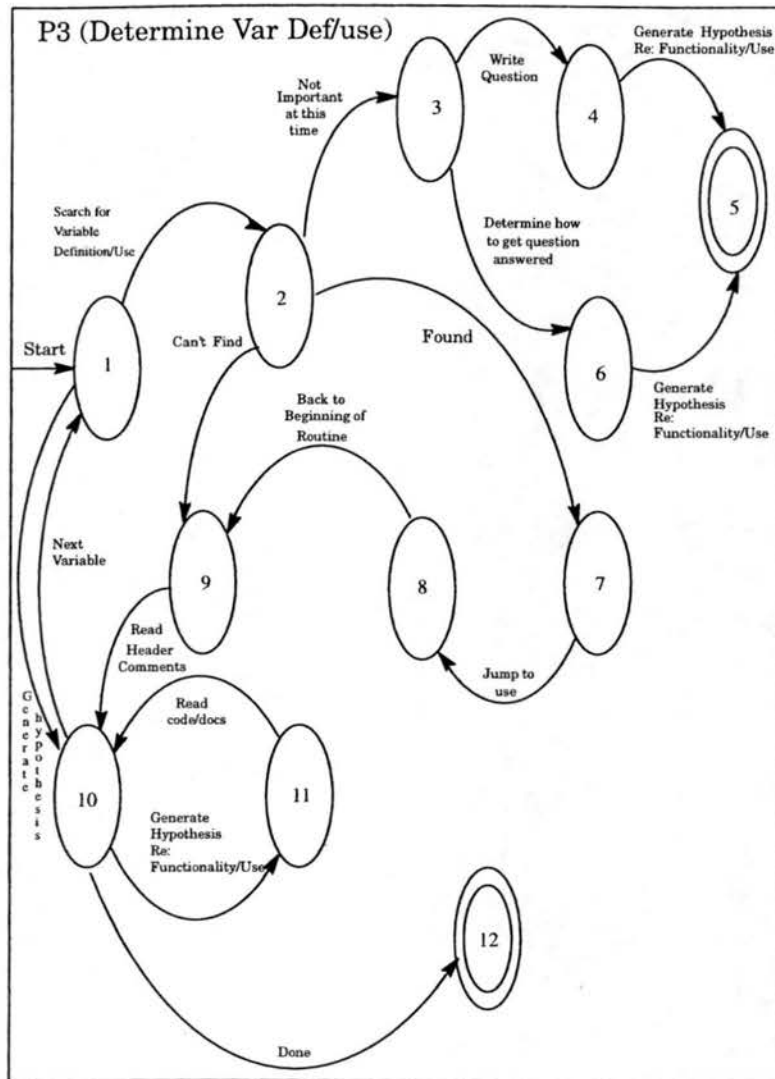


Figure C.7: G2: Process 3 – Determine Var Definition

There are several possible paths through *Determine Variable Definition/Use* process when determining variable definitions or behavior. When this process was found it eventually followed a related *P1* process. This is reasonable since *P1* defers detailed questions about variable definition and use while *P3* answers those questions if possible. The following protocol segment demonstrates a simple path where *G2* saw a reference to variable, determined it was not very important at the moment and deferred resolution by recording a question together with an hypothesis.

But I think what I'll do is look for X3 and see what happens just so I can find out what E is. And uh that looks like it's going to be a lost cause.

**OP6**

Let's go back to 1 and look at X3.incl.pas. So I want to make myself another note here. I need more terminals or something.

**SYS8**

So here's a, it's not exactly a question, but it "Is X3, the X.3 structure defined, we'll have to fix that typo, in X dot, X3.incl.pas? That sounds like that's a reasonable a good hypothesis but we never know for sure.

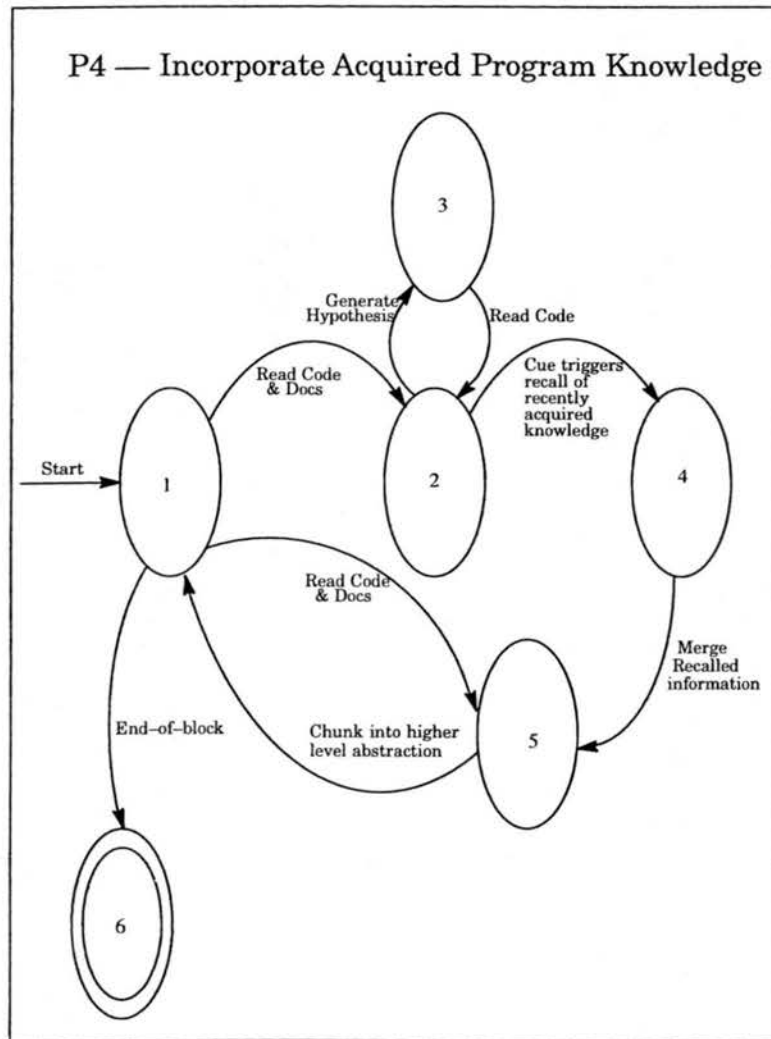


Figure C.8: G2: Process 4 – Incorporate Acquired Knowledge

During the *Incorporate Acquired Knowledge* process, documents or code invoke previously acquired knowledge and he incorporates it into a larger information chunk represented by a higher level of abstraction. The next example protocol chunk illustrates this process.

**SYS1**

And now we're going to look and see, we have a gateway name with an optional X25.

{ Stuff deleted...}

**SYS2, SYS1**

So I need to look for my routine again. Okay, here's my routine. And, okay. Well, there's my comment.

It says, "Length of gateway name ie server with com card with optional X25 physical line number. Name optional as of today."

**SYS11, SYS8**

So I gotta check and see,

it looks like this particular call allows for either a name or a literal address and they're trying to decide which one it is.

**SYS1, Confirmed hypothesis, SYS3**

And that's what the comment has to do with. Optional X25, physical, that's what I thought. I was trying to figure out what PH started.

Okay, so it says if the first character of X25\_NAME,

**Program knowledge, SYS3/SIT1, SYS1** which we um, which we filled in up here —

If the first character is an alphabetic character, that is capital A-Z or little a-z, then we're going to try to resolve the system name to and X.25 address

and it says the way we do that is to look at um, a library of routines called X25\_LIB.PAS

**Program knowledge, SYS8, Confirmed Hypoth, Program knowledge**

and I have looked a little bit at this library and the functions in that appear to be relatively straight forward.

So for the time being I'm going to assume that X25.\$NAME.RESOLVE is fairly straight forward.

In fact it's one that I have looked at

**SYS3/SIT1**

and it basically gets a name and a name length and uh, returns and X25 address and an X25 address length. And also a gateway and a gateway length so that allows it to identify the gateway and it checks to see whether or not it

to identify the gateway and it checks to see whether or not it successfully resolved that name into something. And if it did not successfully resolve it then it writes an error and returns from this function to the calling function.



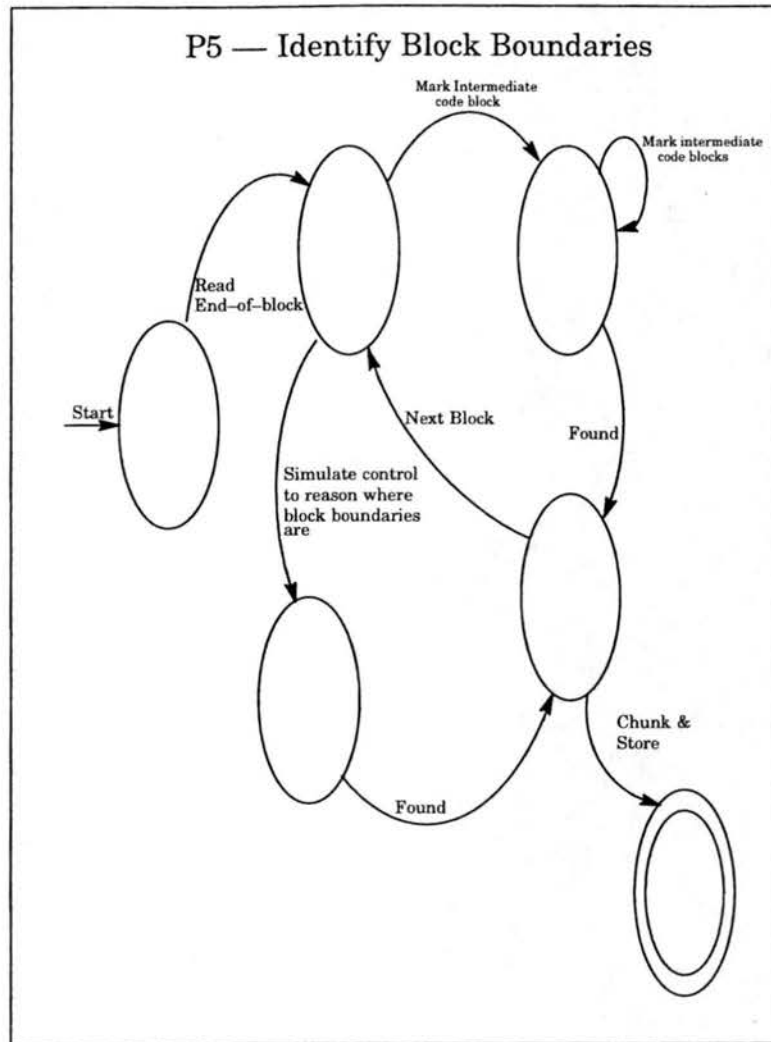


Figure C.9: G2: Process 5 – Identify Block Boundaries

Delineating code blocks is the task embodied with the *Identify Block Boundaries* process. Highlighting begins and ends of code blocks and control simulation are two approaches (paths) seen in the following transcript segment.

**SYS24**

Oh, boy. I may have to go look at the screen to figure what this END is the END of. Yep.

**SYS8**

Because I think it might be the END of this ELSE. But I'm not sure. I wonder if I'm going to be able to figure that out. (ooohhh)

**SYS24**

We're about to find out. Okay, the first thing I need to do is figure out where that END is. And that END is that END right there.

**SYS8**

Wondering if,

**SYS19**

nope. It's not going to let me do it that way. One of the disadvantages of not having this written in C.

{ Stuff deleted... }:

**SYS24** So it certainly lines up with this ELSE. If that's the case then BEGIN, so that's where that one began.

Looks to me like that must be..okay if it's it, let's see what we do to it. Okay. So if it was a literal address..hmmm.

**SYS8/SIT7/SYS21**

What concerns me is that it seems like logically if we had a name that we were trying to resolve into an X.25 address, it looks to me like what we've done here is, resolve the name address, it looks to me like what we've done here is, resolve the name into an X.25 address and then if it was okay, um then what we want to do is jump around this BEGIN, this block starts with ELSE BEGIN and um, it seems to me that we would still want to do the CALL. And so what I'm puzzled over is that if that was a, it looks like to me right here we start on the assumption that the first token was clear or disconnect.

**SYS24**

So if the first token was CALL, which was what we say someplace up here, right there. So here's the start of our block that says CALL.

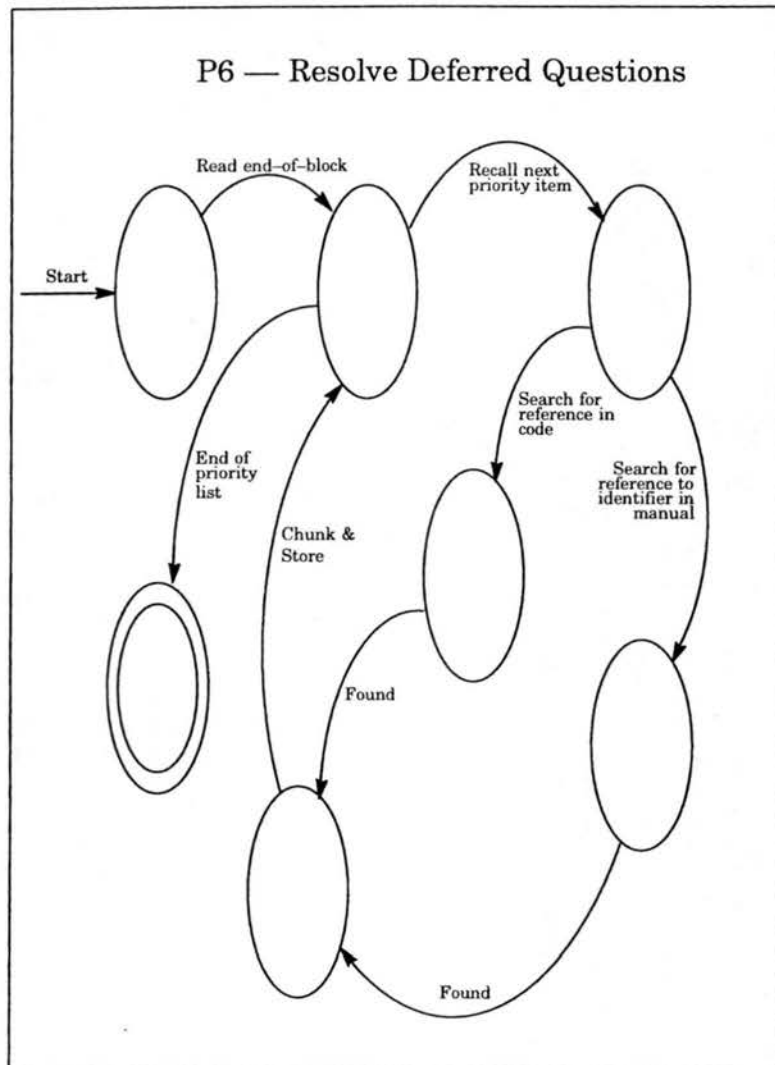


Figure C.10: G2: Process 6 – Resolve Deferred Questions

The *Resolve Deferred Questions* process covers situations when the solution to a previously formulated question is answered intentionally or unintentionally because the programmer stumbles across it while concentrating on unrelated questions. In the transcript fragment G2 finally answers a question regarding the functionality of system procedure called PUT\_REC. Prior to the time represented by the segment, he had been bothered with the use of PUT\_REC. He inadvertently finds the answer while looking for the definition of a variable CLEAR.

**OP1 – Read Manual**

goody, CLEAR\_PACKETS. You pack it. CALL, CALL\_CLEAR, 2-14 and 4-5. Let's see if they tell me anything

particularly enlightening. Okay, Clear, Format Clear, no arguments. Use this command to disconnect the Call from disk commands providing the same function. Okay. And the other was, 4-5. (sigh) 4-5. Call...

I: Are you scanning for page 4-5?

B: Well, it said that it's on page 4-5 that it's referenced. And so I'm looking to see where in here it talks about "CLEAR" and it doesn't tell me...

I: Is is even related? (ha, ha)

#### **SIT1/OP1 – Related knowledge**

B: well, it's a little bit related but it's talking about establishing a channel and we're looking at clearing one. So I'm wondering if closing a connection, it only tells us about CLOSE. It doesn't tell us about CLEAR. Here in exchanging data, it says, after you've established a connection you can begin to exchange data, you send data with PUT\_REC call so in fact PUT\_REC is sending some data across when specifying you must tell it the packet type you are sending. (ah ha!)

*Understand Procedure Call* process is the first of the described processes that invokes another process, *P6*. A sample transcript is not included because only one episode was found in the example protocol and it spans several pages of text. The process begins with the goal of finding a definition of PUT\_REC and includes the P6 protocol segment presented above.

### **Information Needs for Episode-Level Processes**

The knowledge required for successful understanding includes the information needed during comprehension processes. During our analysis we were able to associate information needs with maintenance activities and thus with episode-level processes. Table C.13 contains the information needs associated with processes P1 through P7 for subject G2. Also included are frequencies.

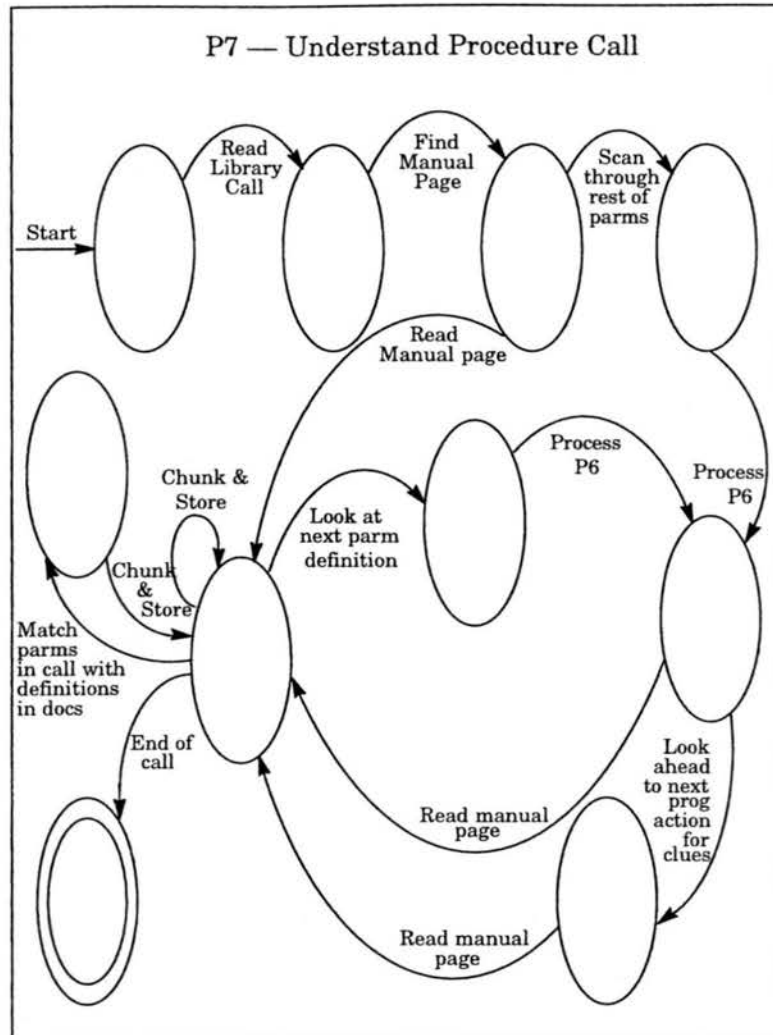


Figure C.11: G2: Process 7 – Understand Procedure Call

Table C.13: G2: Understanding One Module – Information Needs for Processes P1 – P7

<i>Episodic Process</i>	<i>Information Needs</i>	<i>Frequency</i>
P1: Read Block	Code Block Boundaries	4
	Data Type definitions & location of identifiers	3
	Call Graph Display procedure callers	2
	History of past modifications	1
	Data structure definitions tied to concepts in domain	1
	Location of called procedures	1
	History of browsed locations	1
	Beacons tied to situation model or program model	1
	Description of system calls	1
	Location of documents for program & domain	1
P2: Integrate Not Understood	Location of identifier	1
	Include file definitions and locations	1
	High level domain concepts	1
	Description of toggles, switches	1
	History of browsed locations	1
P3: Determine Variable Def/Use	Difference between expected & actual variable use	1
	Constant values and definitions	2
	Location of identifiers	2
	Most frequently used variables	1
	High level domain concepts	1
	Deferred vars/procs during P1	1
	Annotated/Annotatable code	1
	Beacons tied to situation or program model	1
List of experts	1	
P4: Incorporate Acquired Program Knowledge	Prioritized list of identifier references	1
	Code Block Boundaries	2
	Variable/constant definition/use	1
P5: Identify Block Boundaries	Identifiers tied to concepts in domain	1
	Code Block Boundaries with text elision	1
P6: Resolve Deferred Questions	Code Block Boundaries without text elision	1
	List of Deferred variables/procedures	2
	Locations of identifiers	1
	Important domain concepts	1
P7: Understand Procedure Call	Location of domain concept descriptions	1
	Location of description of specific library routine	4
	Naming conventions tied to libraries & to domain concepts	3
	Domain concept descriptions	3
	Variable definitions of parameters in library call	2
	Naming conventions & rules used in naming new procedures	1
	Pre-defined (constant) values & definitions	1
	Description of each parameter & function in specific library routine	1
	To what is a variable initialized & where	1
	Location of domain concept descriptions	1
	Language conventions if different from std language conventions	1
Definition of library routines and uses	1	
System error messages	1	

## Aggregate-level Processes – Understand One Module

Aggregate-level processes are determined using sequences of episode-level processes in protocols. Three aggregate processes were discovered in G2's protocol and we illustrate them below with state diagrams together with a description of the sequences of these aggregate-level processes we observed in the transcript. Triggers cause process changes between the episode-level processes comprising each aggregate process.

These aggregate processes include the following episode-level processes:

1. *PA* : *P1* – Read block in sequence, *P2* – Integrate partial knowledge including information not completely understood, and *P3* – Investigate Variables
2. *PB* : *P1* – Read block in sequence, *P3* – Investigate variables, *P4* – Identify Block Boundaries, and *P5* – Incorporate Acquired Program Knowledge.
3. *PC* : *P1* – Read block in sequence, *P3* – Investigate Variables, *P6* – Resolve Deferred Questions, and *P7* – Understand Library Procedure Call.

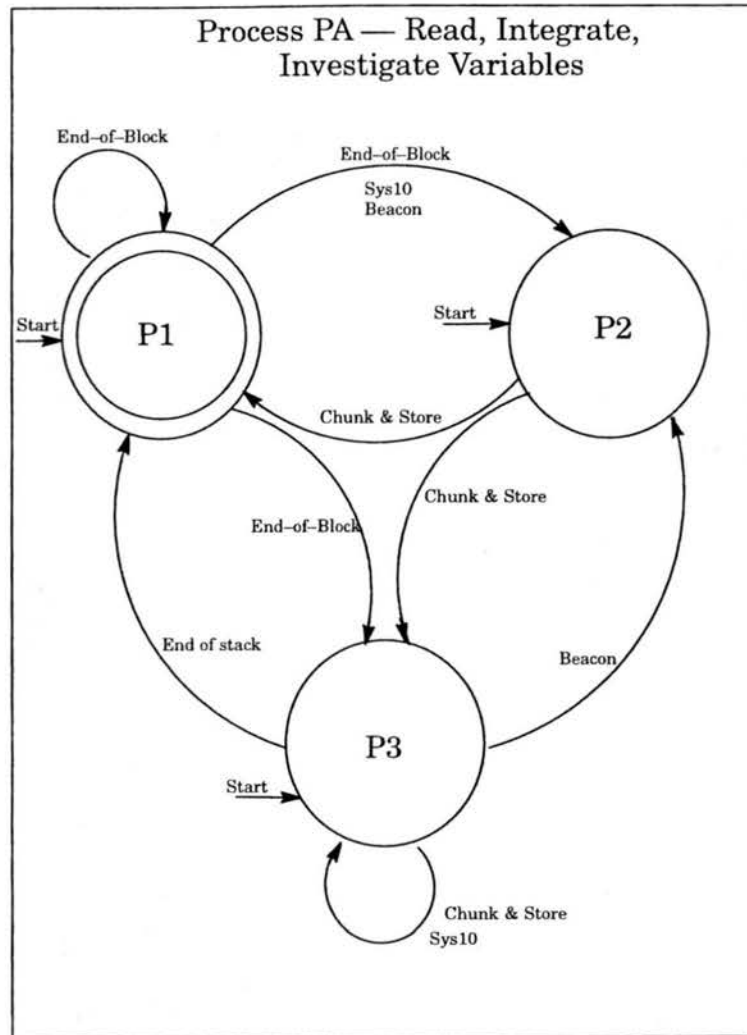


Figure C.12: G2: Process A – Aggregate-Level

The following paths through Figure C.12 were found:

$P1 \rightarrow P1 \rightarrow P2 \rightarrow P1 \rightarrow P3 \rightarrow P3 \rightarrow P2 \rightarrow P3 \rightarrow P3 \rightarrow P3 \rightarrow P1$

$P1 \rightarrow P2 \rightarrow P1$

$P1 \rightarrow P1 \rightarrow P2 \rightarrow P1$

$P1 \rightarrow P3 \rightarrow P3 \rightarrow P1$

Process *PA* is a fully connected state diagram with self loops for episode-level processes *P1* and *P3* only. An intuitive description of this aggregate process describes typical



progressions of episodes as follows:

As code is read, identifiers such as variables and procedure calls are prioritized in the order they will be investigated after reaching the end of the block (*P1*). If there is an unexpected use of an identifier or combination of statements (i.e. rules of discourse are violated) then the confusing information is somehow incorporated into the current working mental representation of the code (*P2*). Once a logical block of code is read and all relevant detail is arranged in a mental model, the priority list containing the deferred identifier information can be examined (*P3*).

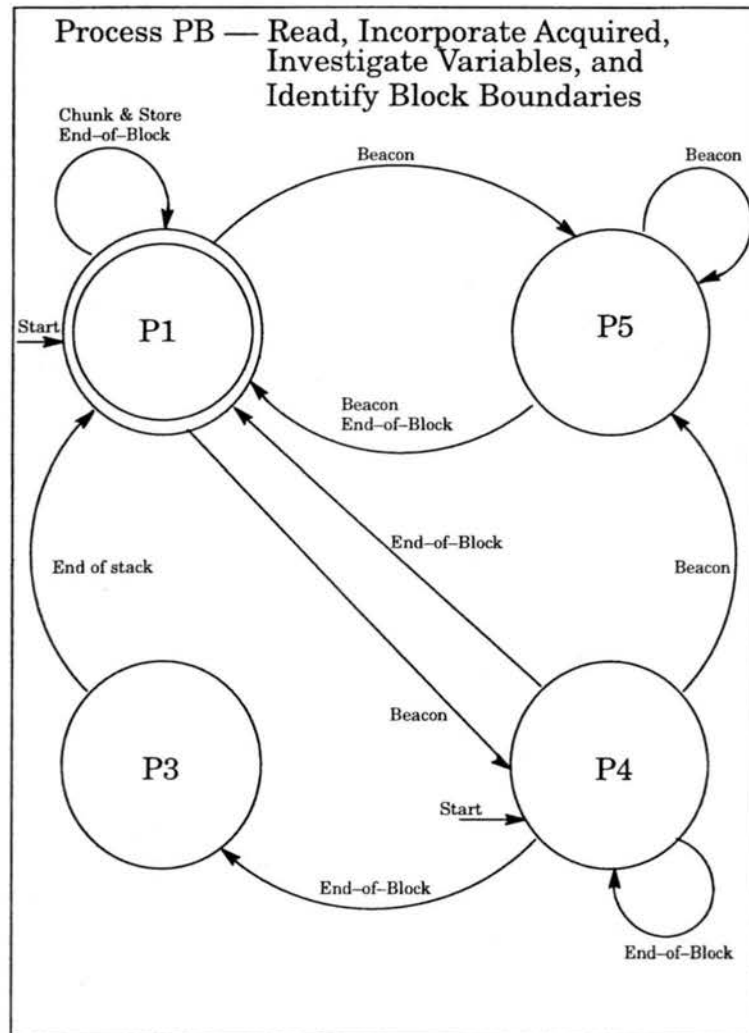


Figure C.13: G2: Process B – Aggregate-Level

The following paths through Figure C.13 include:

$$P1 \rightarrow P5 \rightarrow P1 \rightarrow P4 \rightarrow P1$$
$$P1 \rightarrow P4 \rightarrow P4 \rightarrow P3 \rightarrow P1 \rightarrow P1$$
$$P1 \rightarrow P4 \rightarrow P5 \rightarrow P5 \rightarrow P1$$

Process *PB* is different from *PA* because the presence of the two episode-level processes *Incorporate Acquired Knowledge* (*P4*) and *Identify Block Boundaries* (*P5*) suggest some accumulated knowledge and larger code blocks, respectively. The first occurrence of *PB* was observed at the beginning of the twelfth episode in the protocol. By this time G2 had considerable knowledge about the program. He was also several nesting levels deep in the control structure. He was uncomfortable with being unsure of the control level he was currently examining and decided to explicitly locate the physical block boundaries.

The following paths through Figure C.14 include:

$$P1 \rightarrow P7 \rightarrow P7 \rightarrow P1$$
$$P1 \rightarrow P7 \rightarrow P7 \rightarrow P3 \rightarrow P7 \rightarrow P6 \rightarrow P7 \rightarrow P6 \rightarrow P6 \rightarrow P7 \rightarrow P7 \rightarrow P1$$
$$P1 \rightarrow P1 \rightarrow P7 \rightarrow P7 \rightarrow P1$$

Process *PC* includes resolving deferred questions *P6* and understanding procedure calls *P7* episode-level processes. It is similar to *PA* except that understanding procedure calls are more involved and may require additional documentation such as maintenance or reference manuals. In addition, *PC* is observed only after enough knowledge about the code is accumulated to cause resolution of deferred questions. Our subject actually invoked *P6* when he stumbled across the answer to an unresolved question while looking for clues to an unrelated problem.

### Aggregate-Level Triggers

Triggers are important causes of state changes between episode-level processes. Table C.14 lists triggers found in the example protocol and their frequencies for processes *A*, *B*, *C*, together with a total for all the aggregate processes.

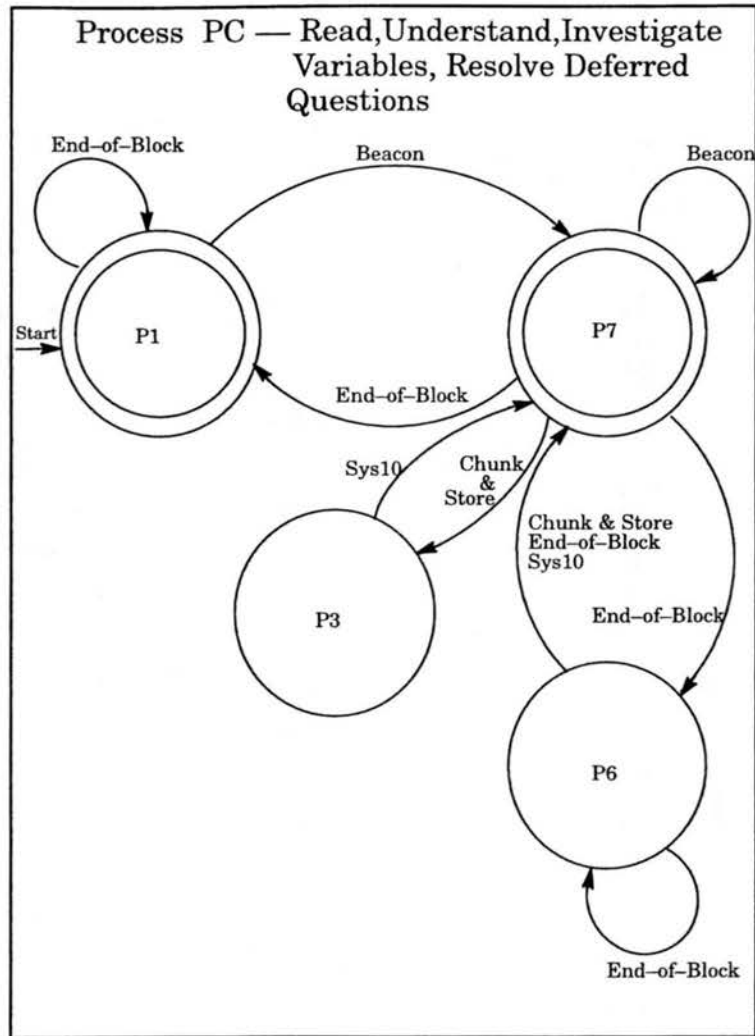


Figure C.14: G2: Process C – Aggregate-Level

Table C.14: G2: Processes A,B,C, and Aggregate Trigger Frequencies

<i>Process Id</i>	<i>Process Trigger</i>	<i>Frequency</i>
PA	Beacon	2
	Chunk & Store	5
	End-of-Block	7
	End-of-Stack	1
	Sys10	5
PB	Beacon	5
	Chunk & Store	1
	End-of-Block	6
	End-of-Stack	1
	Sys10	0
PC	Beacon	7
	Chunk & Store	2
	End-of-Block	7
	End-of-Stack	0
	Sys10	2
All Aggregate Processes	Beacon	14
	Chunk & Store	8
	End-of-Block	20
	End-of-Stack	2
	Sys10	7

## Session Level Process – Understand One Module

Session-level processes (for Understanding a single module) are interesting because they show that all the aggregate-level processes represent investigation towards building chunks [36, 45]. Chunking is an important abstraction mechanism in code understanding. Thus, at the session-level the purpose of each aggregate process is to understand a block of code (using different detail steps and information) and then to chunk and store the learned information.

Session level processes comprise the highest level process encompassing the entire maintenance task for the two hour programming session. The state diagram in Figure C.15 was derived in the same way as the aggregate-level processes by tracking the sequences of aggregate-level processes. At the highest level, only End-of-block and Chunk & Store trigger switches from one aggregate-level process to the next.

The following paths through Figure C.15 were found:

$$PA \rightarrow PB \rightarrow PC \rightarrow PA$$
$$PA \rightarrow PB \rightarrow PA$$
$$PA \rightarrow PC \rightarrow PA$$
$$PA \rightarrow PC \rightarrow PB$$

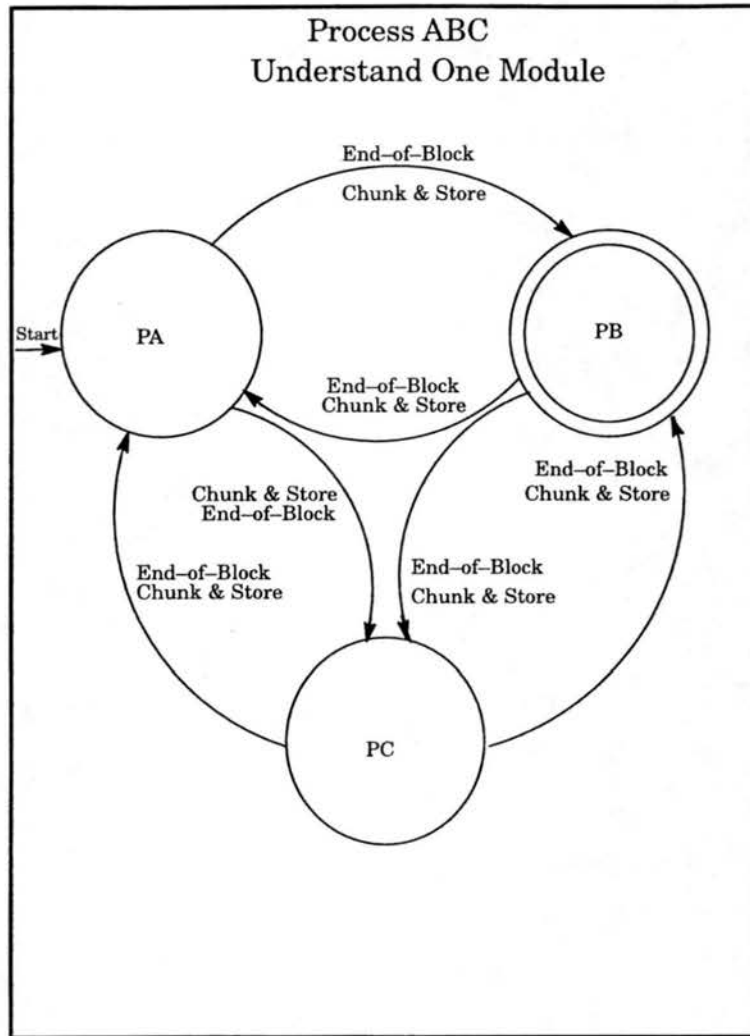


Figure C.15: G2: Process ABC – Session Level

## C.2 Corrective Maintenance

Four subjects worked on corrective maintenance tasks, C1, C2, C3 and C4.

### C.2.1 C1: Domain Novice/Language Expert – Fix Reported Bug

C1's task was to fix a bug that had been reported by a customer. He is an expert Pascal programmer but a novice in the application domain. The domain is terminal emulation software which includes communications protocols. The documentation he had access to included a bug report, an internal maintenance manual, a text book on communications networks, and a history of previous bug fixes. He had spent only two hours prior to the programming session looking over the bug reports and staring at the block of code identified in the bug report as the problem. He had also spent some time discussing the bug with the engineer that had previous responsibility for the code. He had a copy of the code on-line which he used specifically for searching. A custom cross referencing tool found all instances of specific identifiers.

At the beginning of the programming session, C1 felt he knew which statement contained the bug and concentrated solely on that module. His approach to understanding was opportunistic in that he concentrated on the areas considered relevant. This turned out to be detrimental because after two hours, he was no closer to understanding the bug. Instead, he had only determined that his original hypothesis about the location of the bug was incorrect and now he needed to locate the bug.

Table C.15: C1: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
49	Top-Down Model	N/A	14	14
162	Situation Model	18	N/A	29
145	Program Model	11	32	N/A

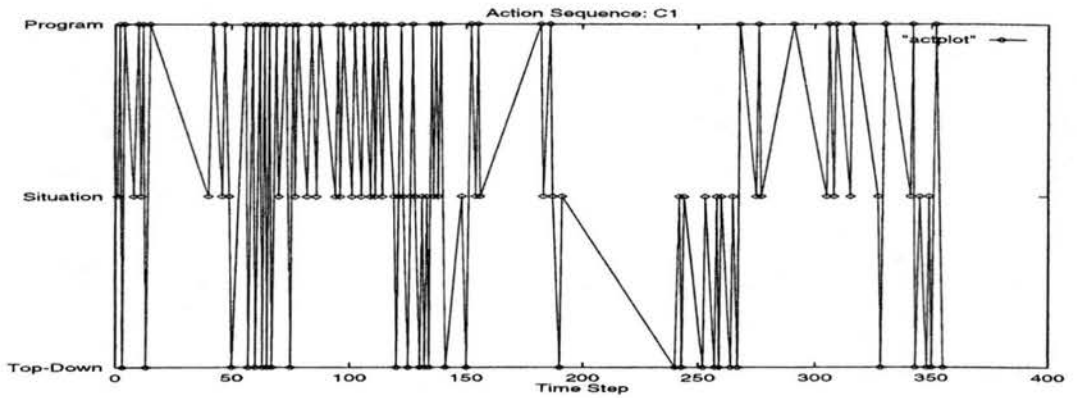


Figure C.16: C1: Fix Reported Bug – Action Sequence

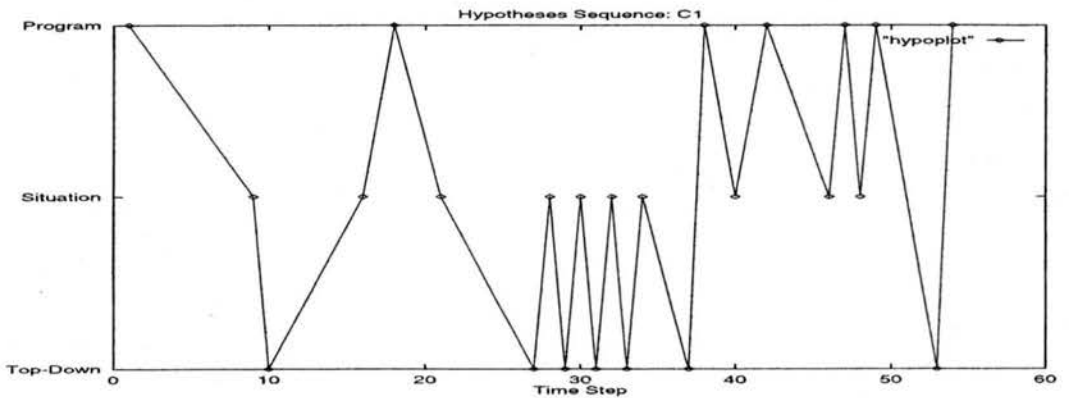


Figure C.17: C1: Fix Reported Bug – Hypotheses Sequence



Table C.16: C1: Action-Type Frequencies – Fix Reported Bug

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	6	13%
OP2	Determine next prgm. seg to examine	7	14%
OP3	Generate/revise hypothesis re: functionality	9	18%
OP4	Determine relevance of prgrm segment	1	2%
OP6	Determine understand strategy	8	16%
OP8	Failed hypothesis	2	4%
OP15	Generate questions	1	2%
OP17	Chunk & store knowledge	4	8%
OP20	Generate task	4	8%
OPCONF	Confirmed hypothesis	1	2%
OPKNOW	Top-down knowledge	6	13%
<i>Total</i>	<i>Top-Down Model Actions</i>	49	14%
SIT1	Gain situation knowledge	38	23%
SIT2	Develop questions	13	8%
SIT3	Determine answers to questions	6	4%
SIT4	Chunk & store	41	25%
SIT5	Determine relevance of sit. know.	5	3%
SIT6	Determine next info to gain	6	4%
SIT7	Generate hypothesis	17	10%
SIT8	Determine understand strategy	5	3%
SIT10	Failed hypothesis	1	1%
SITCONF	Confirmed hypothesis	1	1%
SITKNOW	Situation model knowledge	29	18%
<i>Total</i>	<i>Situation Model Actions</i>	162	45%
SYS1	Read intro code comments/related docs	4	3%
SYS2	Determine next prg segmt to examine	15	10%
SYS3	Examine next module in sequence	21	15%
SYS4	Examine next module in cntrl-flow	1	1%
SYS5	Examine data structs & definitions	4	3%
SYS7	Chunk & store knowledge	21	14%
SYS8	Generate hypothesis	24	17%
SYS9	Construct call tree	3	2%
SYS10	Determine understand strategy	19	13%
SYS19	Failed hypothesis	2	1%
SYS21	Mental simulation	5	3%
SYS24	Search for block begin/end	2	1%
SYSKNOW	Program model knowledge	24	17%
<i>Total</i>	<i>Program Down Model Actions</i>	145	41%

Table C.17: C1: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
9	Top-Down Model	N/A	5	2
17	Situation Model	6	N/A	4
24	Program Model	1	5	N/A

Table C.18: C1: Hypothesis-Type Frequencies – Fix Reported Bug

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	1	11%
	OPH2	Variable functionality/domain concepts	1	11%
	OPH13	Number/type/location of file	1	11%
	OPH16	Level & structure of code/scope	5	56%
	OPH18	Location/Status/description/cause of error	1	11%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	9	18%
Situation Model	SITH1	Variable function	2	12%
	SITH2	Function/code block execution order/state	1	6%
	SITH3	Function/procedure function, call function	8	47%
	SITH5	Cause of buggy behavior	1	6%
	SITH6	Comparison of terms/acronyms /functionality	3	17%
	SITH7	Existence of functionality/algorithm/variable	2	12%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	17	34%
Program Model	SYSH6	Statement execution order/state	11	46%
	SYSH7	Variable value/defaults	1	4%
	SYSH10	Syntax meaning	5	21%
	SYSH13	Code block/procedure comparison	2	8%
	SYSH16	Code correctness, cause/location of error	4	17%
	SYSH18	Location to add code/alternatives	1	4%
	<i>Total</i>	<i>Program Model Hypotheses</i>	24	48%

Table C.19: C1: Fix Reported Bug - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	14
I61	Connected domain-program-situation model knowledge	SYS7	5
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	3
I9	List of browsed locations	OP2, OP4, SYS2 OP4	2
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	4
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	4
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	7
I3	Highlighted begin/ends of control blocks	SYS3, SYS7, SYS8, SYS10, SYS24	2
I22	History of past modifications	SYS3	3
I73	Bug behavior isolated	SIT1	1
I14	Call Graph Display	OP1, OP2, SYS3	3
I68	List of issues/decisions considered during design	SYS12	1
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	1
I58	High-level description of how code is laid out	OP3	1
I72	Good direction to follow given what is already known, possible program segments to examine	SYS10	1
I66	Expected program state, e.g. expected variable values when procedure is called	SYS11	2
I74	Good description of the bug and why and how other fixes were done	SIT1	1
I11	Language conventions that are different from standard conventions Language enhancements	SYS1, SYS3, SYS10	2
I62	Predefined (constant) variables and values	SYS8	1
I13	Conditions under which a branch is taken or not. Include variable values	SYS3	4
I30	Where variable is toggled, when and why, where passed to and why	SYS2, SYS12	2
I38	Nesting level of a particular procedure	SYS2, SYS7	5
I19	Acronym definitions	SIT1, SIT7	3

### C.2.2 C2: Domain Expert/Language Novice – Understand Bug

C2's task was to understand a reported bug. The program he worked on had the capability for processing command line input, however, it was handling inputs incorrectly. The program was a software management tool that allows users to create PERT charts and GANT charts. Users normally interact with the program through an X-Windows user interface. For batch processing a well behaved command line interface was necessary. The interface code was implemented using the X-Windows Tool Kit (XT Intrinsic), with which C2 was unfamiliar. He had eight years of Microsoft Windows and C programming experience. Thus, he was an event-driven application expert, but a Unix XT Toolkit novice. At the time of the programming session, he had never looked at the code or any supporting documentation. His basic strategy was to start with the main program and understand the event loop so could determine how the command line interface interacted with the event loop.

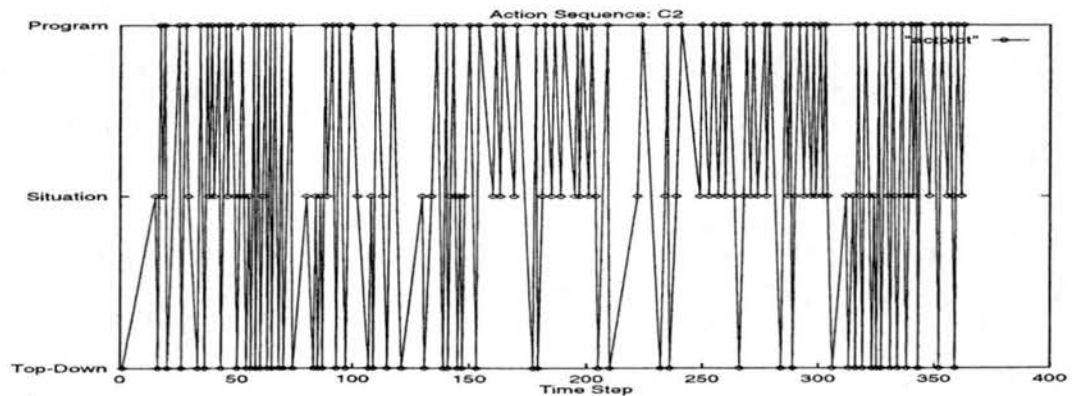


Figure C.18: C2: Understand Bug – Action Sequence

Table C.20: C2: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
120	Top-Down Model	N/A	23	31
92	Situation Model	26	N/A	41
151	Program Model	27	44	N/A

Table C.21: C2: Action-Type Frequencies – Understand Bug

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	16	13%
OP2	Determine next prgm. seg to examine	3	2%
OP3	Generate/revise hypothesis re: functionality	18	15%
OP4	Determine relevance of prgm segment	8	7%
OP6	Determine understand strategy	12	10%
OP7	Investigate oversight	1	1%
OP8	Failed hypothesis	1	1%
OP11	High-level change plan/alternatives	7	6%
OP13	Study/initiate program execution	3	2%
OP14	Compare program segments	2	2%
OP15	Generate questions	4	3%
OP16	Answer questions	1	1%
OP17	Chunk & store knowledge	9	8%
OP18	Change directions	2	2%
OP20	Generate task	9	7%
OPKNOW	Top-down knowledge	24	20%
<i>Total</i>	<i>Top-Down Model Actions</i>	120	33%
SIT1	Gain situation knowledge	10	11%
SIT2	Develop questions	4	4%
SIT3	Determine answers to questions	1	1%
SIT4	Chunk & store	29	32%
SIT5	Determine relevance of sit. know.	4	4%
SIT6	Determine next info to gain	2	2%
SIT7	Generate hypothesis	18	20%
SIT8	Determine understand strategy	2	2%
SIT11	Mental simulation	2	2%
SIT12	Compare functionality of 2 versions	2	2%
SITCONF	Confirmed hypothesis	1	1%
SITKNOW	Situation model knowledge	17	19%
<i>Total</i>	<i>Situation Model Actions</i>	92	25%
SYS1	Read intro code comments/related docs	15	10%
SYS2	Determine next prg segmt to examine	7	5%
SYS3	Examine next module in sequence	29	19%
SYS4	Examine next module in cntrl-flow	2	1%
SYS7	Chunk & store knowledge	23	15%
SYS8	Generate hypothesis	12	8%
SYS10	Determine understand strategy	14	9%
SYS11	Generate new task	13	9%
SYS12	Generate question	9	6%
SYS13	Determine if looking at right code	3	2%
SYS14	Change direction	1	1%
SYS15	Generate/consider different code changes	13	9%
SYS17	Add/Alter code	2	1%
SYS19	Failed hypothesis	2	1%
SYS23	Search for var defines/use	1	1%
SYSKNOW	Program model knowledge	5	3%
<i>Total</i>	<i>Program Down Model Actions</i>	151	42%

Table C.22: C2: Hypothesis-Type Frequencies – Understand Bug

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of TU</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	3	17%
	OPH2	Variable functionality/domain concepts	1	5%
	OPH6	Existence of specific functionality	3	17%
	OPH7	Number/type/existence/location of libraries	4	22%
	OPH9	Permissions/Environment set correctly/ Tool functionality	2	11%
	OPH11	Comparison of functionality at high level	1	6%
	OPH13	Number/type/location of file	2	11%
	OPH17	Design Decisions/Modifications	2	11%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	18	37%
Situation Model	SITH2	Function/code block execution order/state	6	33%
	SITH3	Function/procedure function, call function	7	39%
	SITH6	Comparison of terms/acronyms /functionality	1	6%
	SITH7	Existence of functionality/algorithm/ variable	4	22%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	18	38%
Program Model	SYSH1	Variable function	1	8%
	SYSH2	Function/procedure function	2	17%
	SYSH4	Variable structure	4	34%
	SYSH6	Statement execution order/state	1	8%
	SYSH8	(Non-)Existence of construct (var/code)	2	17%
	SYSH9	Variable/construct equivalency	1	8%
	SYSH18	Location to add code/alternatives	1	8%
	<i>Total</i>	<i>Program Model Hypotheses</i>	12	25%

Table C.23: C2: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
18	Top-Down Model	N/A	5	6
18	Situation Model	8	N/A	5
12	Program Model	2	9	N/A

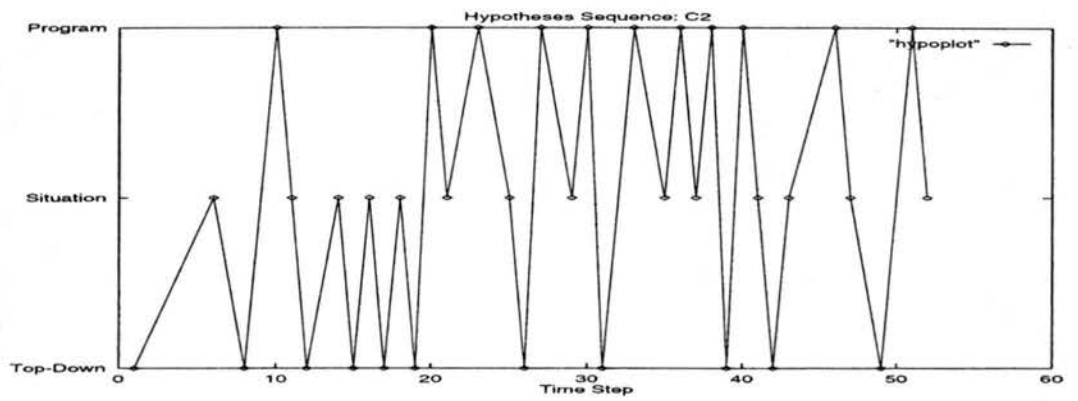


Figure C.19: C2: Fix Reported Bug – Hypotheses Sequence



Table C.24: C2: Fix Reported Bug - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	18
I61	Connected domain-program-situation model knowledge	SYS7	12
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	8
I9	List of browsed locations	OP2, OP4, SYS2 OP4	7
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	1
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	6
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	1
I22	History of past modifications	SYS3	2
I14	Call Graph Display	OP1, OP2, SYS3	4
I68	List of issues/decisions considered during design	SYS12	1
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	1
I17	Location of desired code segment	SYS1, SYS2	1
I6	Location and description of library routines and system calls	OP1, OP2, OP20, SYS11	2
I58	High-level description of how code is laid out	OP3	3
I20	Documentation list and location	OP1	2
I44	List of routines that do most of the domain-type work	OP3	4
I72	Good direction to follow given what is already known, possible program segments to examine	SYS10	3
I16	Naming conventions separated by system or library objects that use them Rules used for new naming new procedures	OP3, SYS3, SYS8 SIT7	4
I12	Environment, global, local scope	SYS3	1
I62	Predefined (constant) variables and values	SYS8	1
I39	Main program location	OP1	3
I47	List of identifiers and domain concepts that are important	OP4	1
I49	File name of current file	SYS3	1
I55	Domain Simulation	SIT2	1
I21	Organized functions into categories in which functions are related	OP2	1
I26	All include file definitions and uses	OP1, SYS5	1
I28	List of all routines with initialization code	OP2, SYS2	1
I8	List of system calls based on specific naming convention	OP1	2

### C.2.3 C3: Domain Expert/Language Novice – Fix Reported Bug

C3's task involved fixing a defect in the operating system's kernel. Prior to the programming session, he had spent a couple of days reproducing the bug and stepping through the program using a debugger. This gave him a high-level understanding of the code structure and the particular code modules on which he should focus his attention. C3 was an operating systems expert. He had worked for the previous five years on Unix operating system development. He was a C programming language expert with 8 years of experience. The system he was working with was written in Pascal, a language he had worked with for less than a year.

The strategy he took to finding the defect was to concentrate on the modules he had previously identified as related to the bug. He used an internally developed tool for finding cross references to identifiers. For key data structures, he manually constructed data-flow representations using Emacs. At the end of the session, he had determined that the defect was not manifested in the area of the program where he had originally assumed it was located.

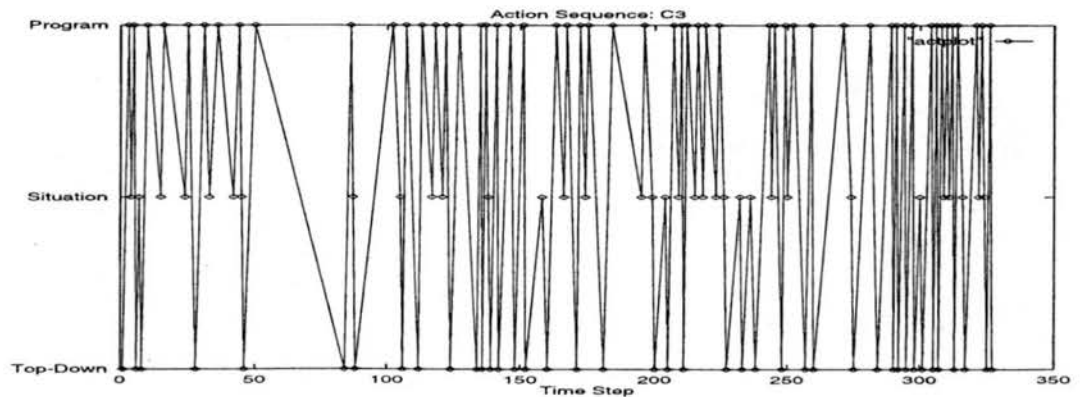


Figure C.20: C3: Fix Reported Bug – Action Sequence

Table C.25: C3: Action-Type Frequencies – Fix Reported Bug

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP2	Determine next prgm. seg to examine	23	19%
OP3	Generate/revise hypothesis re: functionality	25	21%
OP4	Determine relevance of prgrm segment	1	1%
OP6	Determine understand strategy	14	11%
OP8	Failed hypothesis	1	1%
OP14	Compare program segments	1	1%
OP15	Generate questions	4	3%
OP16	Answer questions	2	2%
OP17	Chunk & store knowledge	11	9%
OPCONF	Confirmed hypothesis	1	1%
OPKNOW	Top-down knowledge	38	31%
<i>Total</i>	<i>Top-Down Model Actions</i>	121	37%
SIT4	Chunk & store	13	31%
SIT5	Determine relevance of sit. know.	2	5%
SIT7	Generate hypothesis	7	16%
SIT8	Determine understand strategy	2	5%
SITKNOW	Situation model knowledge	18	43%
<i>Total</i>	<i>Situation Model Actions</i>	42	13%
SYS1	Read intro code comments/related docs	7	4%
SYS2	Determine next prg segmt to examine	3	2%
SYS3	Examine next module in sequence	43	26%
SYS7	Chunk & store knowledge	51	30%
SYS8	Generate hypothesis	18	11%
SYS10	Determine understand strategy	9	5%
SYS12	Generate question	1	1%
SYS16	Answer question	1	1%
SYS19	Failed hypothesis	2	1%
SYS21	Mental simulation	3	2%
SYS23	Search for var defines/use	3	2%
SYSCONF	Confirmed hypothesis	4	2%
SYSKNOW	Program model knowledge	21	13%
<i>Total</i>	<i>Program Down Model Actions</i>	166	50%

Table C.26: C3: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
121	Top-Down Model	N/A	6	34
42	Situation Model	15	N/A	19
166	Program Model	25	28	N/A

Table C.27: C3: Hypothesis-Type Frequencies – Fix Reported Bug

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	5	19%
	OPH3	Rules of discourse/Expectations	1	4%
	OPH7	Number/type/existence/location of libraries	1	4%
	OPH11	Comparison of functionality at high level	1	4%
	OPH13	Number/type/location of file	1	4%
	OPH14	Available functionality	2	7%
	OPH16	Level & structure of code/scope	6	23%
	OPH18	Location/Status/description/cause of error	8	31%
	OPH19	Current location	1	4%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	26	51%
Situation Model	SITH2	Function/code block execution order/state	1	14%
	SITH3	Function/procedure function, call function	3	43%
	SITH5	Cause of buggy behavior	2	29%
	SITH7	Existence of functionality/algorithm/variable	1	14%
		<i>Total</i>	<i>Situation Model Hypotheses</i>	7
Program Model	SYSH2	Function/procedure function	3	17%
	SYSH5	Location/type/existence of function call	1	6%
	SYSH6	Statement execution order/state	2	11%
	SYSH7	Variable value/defaults	2	11%
	SYSH10	Syntax meaning	2	11%
	SYSH16	Code correctness, cause/location of error	6	33%
	SYSH20	Params/type definitions in procedure call	2	11%
	<i>Total</i>	<i>Program Model Hypotheses</i>	18	35%

Table C.28: C3: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
26	Top-Down Model	N/A	2	6
7	Situation Model	2	N/A	3
18	Program Model	7	2	N/A

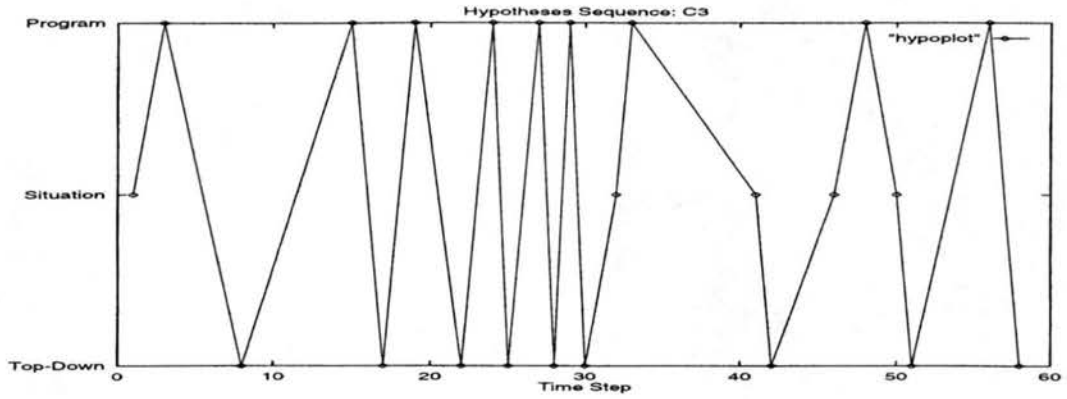


Figure C.21: C3: Fix Reported Bug - Hypotheses Sequence

Table C.29: C3: Fix Reported Bug - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	2
I61	Connected domain-program-situation model knowledge	SYS7	10
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	8
I9	List of browsed locations	OP2, OP4, SYS2 OP4	8
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	3
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	1
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	6
I73	Bug behavior isolated	SIT1	5
I14	Call Graph Display	OP1, OP2, SYS3	5
I6	Location and description of library routines and system calls	OP1, OP2, OP20, SYS11	1
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	1
I20	Documentation list and location	OP1	1
I44	List of routines that do most of the domain-type work	OP3	1
I72	Good direction to follow given what is already known, possible program segments to examine	SYS10	2
I36	Sequence of locations where ID is used	OP2	5
I62	Predefined (constant) variables and values	SYS8	1
I41	Call graph with extraneous information not relevant elided	OP2	2
I37	Language definitions, e.g. reserved words, instruction defs, for C, Pascal, etc.	SYS8, SYS18	1
I49	File name of current file	SYS3	1
I10	Function call count	OP1, SYS2	1
I21	Organized functions into categories in which functions are related	OP2	1
I63	All definitions and uses of a variable prioritized. so more important uses show up first	SYS10	2
I59	How a variable is passed into a procedure, e.g. by value, address, etc.	SYS1	1

#### **C.2.4 C4: Domain Expert/Language Novice – Track Down Bug**

The problem C4 worked on involved the loss of data between two Unix sockets when a server sends data to a client program. Prior to the programming session he had worked on the code to fix two additional defects. C4 is an operating systems expert with ten to eleven years of experience on Unix operating system development. He is also an expert C programmer. The system was written in Pascal, a language he had only been working with for seven months.

He had already reproduced the defect by loading the program into a debugger and watching the contents of the data structure passed between the server and client. His strategy for locating the defect in the code was to re-load the program into a debugger and step through the program and the associated system library functions. At the point where he thought the defect was, he read code line by line. When this strategy failed, he stepped over code he did not feel was relevant and searched for areas that had been recently changed by previous owners of the code. At the end of the session, he had stepped through all code on the server side and did not locate the defect. He indicated that his next strategy would be to load the client code into the debugger and step through it in a manner similar to the way he looked at the server side.

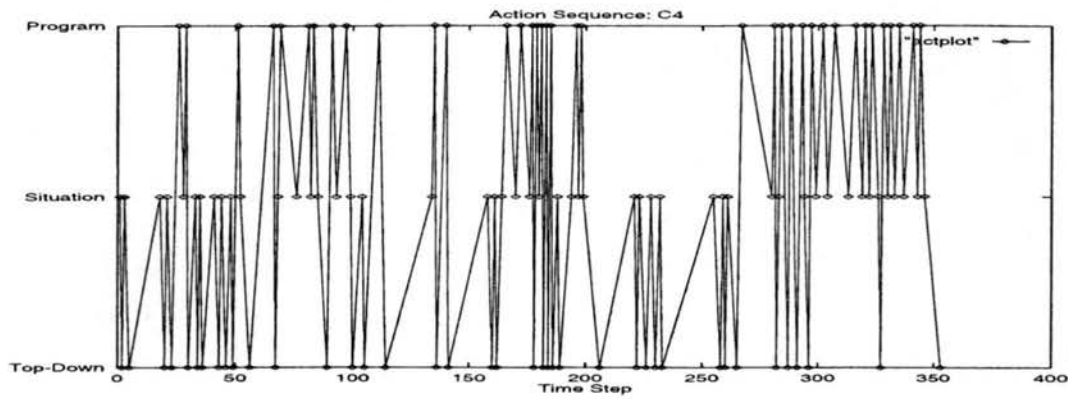


Figure C.22: C4: Track Down Bug – Action Sequence

Table C.30: C4: References and Action Switches Between Models

Number of References	Model	Model Switches – Understand Program		
		Top-Down Model	Situation Model	Program Model
171	Top-Down Model	N/A	24	14
103	Situation Model	27	N/A	23
89	Program Model	12	25	N/A



Table C.31: C4: Action-Type Frequencies – Track Down Bug

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	5	3%
OP2	Determine next prgm. seg to examine	12	7%
OP3	Generate/revise hypothesis re: functionality	24	14%
OP4	Determine relevance of prgrm segment	8	4%
OP5	Determine if prgm seg needs detail understand	1	1%
OP6	Determine understand strategy	9	5%
OP7	Investigate oversight	1	1%
OP9	Mental simulation	1	1%
OP11	High-level change plan/alternatives	1	1%
OP12	Observe buggy behavior	3	2%
OP13	Study/initiate program execution	20	11%
OP15	Generate questions	5	3%
OP16	Answer questions	1	1%
OP17	Chunk & store knowledge	5	3%
OP20	Generate task	36	21%
OPKNOW	Top-down knowledge	22	39%
<i>Total</i>	<i>Top-Down Model Actions</i>	171	47%
SIT1	Gain situation knowledge	1	1%
SIT2	Develop questions	7	7%
SIT3	Determine answers to questions	1	1%
SIT4	Chunk & store	24	23%
SIT5	Determine relevance of sit. know.	4	4%
SIT6	Determine next info to gain	7	7%
SIT7	Generate hypothesis	22	21%
SIT8	Determine understand strategy	5	5%
SIT10	Failed hypothesis	1	1%
SIT11	Mental simulation	2	2%
SITCONF	Confirmed hypothesis	1	1%
SITKNOW	Situation model knowledge	28	27%
<i>Total</i>	<i>Situation Model Actions</i>	103	28%
SYS1	Read intro code comments/related docs	1	1%
SYS2	Determine next prg segmt to examine	1	1%
SYS3	Examine next module in sequence	9	11%
SYS4	Examine next module in cntrl-flow	13	15%
SYS5	Examine data structs & definitions	1	1%
SYS7	Chunk & store knowledge	20	23%
SYS8	Generate hypothesis	11	12%
SYS10	Determine understand strategy	4	4%
SYS11	Generate new task	8	9%
SYS12	Generate question	1	1%
SYS13	Determine if looking at right code	4	5%
SYS19	Failed hypothesis	3	3%
SYS21	Mental simulation	1	1%
SYSCONF	Confirmed hypothesis	1	1%
SYSKNOW	Program model knowledge	11	12%
<i>Total</i>	<i>Program Down Model Actions</i>	89	25%

Table C.32: C4: Hypothesis-Type Frequencies – Track Down Bug

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Tot</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	2	8%
	OPH8	Program functions correctly	1	4%
	OPH9	Permissions/Environment set correctly/ Tool functionality	9	38%
	OPH13	Number/type/location of file	1	4%
	OPH18	Location/Status/description/cause of error	11	46%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	24	42%
Situation Model	SITH1	Variable function	5	23%
	SITH2	Function/code block execution order/state	1	5%
	SITH3	Function/procedure function, call function	7	32%
	SITH4	Effect of running program	1	5%
	SITH5	Cause of buggy behavior	3	13%
	SITH7	Existence of functionality/algorithm/ variable	3	13%
	SITH8	Program function	2	9%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	22	39%
Program Model	SYSH2	Function/procedure function	1	9%
	SYSH4	Variable structure	3	27%
	SYSH6	Statement execution order/state	1	9%
	SYSH7	Variable value/defaults	5	46%
	SYSH8	(Non-)Existence of construct (var/code)	1	9%
	<i>Total</i>	<i>Program Model Hypotheses</i>	11	19%

Table C.33: C4: Hypothesis Generated Switches Between Models

Number of Hypotheses	Model	Model Switches - Understand Program		
		Top-Down Model	Situation Model	Program Model
24	Top-Down Model	N/A	7	2
22	Situation Model	7	N/A	5
11	Program Model	2	5	N/A

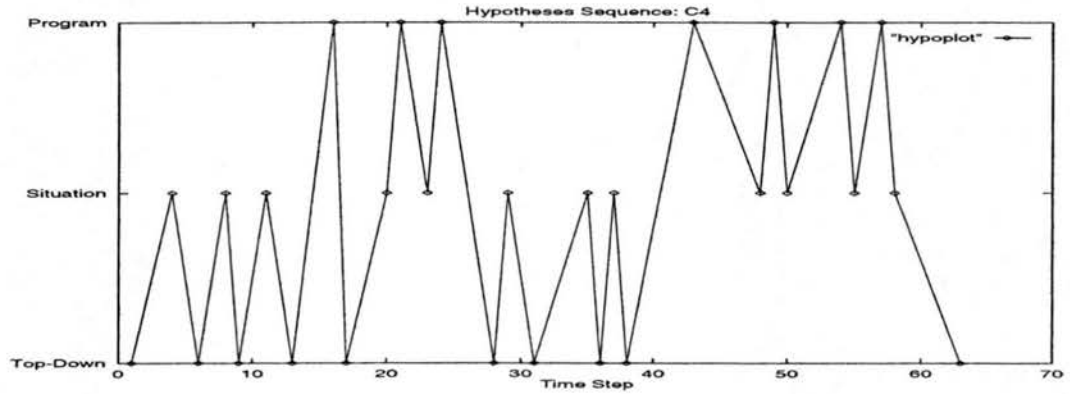


Figure C.23: C4: Track Down Bug – Hypotheses Sequence

Table C.34: C4: Track Down Bug - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
17	Domain concept descriptions	OP1	7
I61	Connected domain-program-situation model knowledge	SYS7	6
14	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	7
11	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10, SYS12, SYS17, SYS23	2
I22	History of past modifications	SYS3	2
I73	Bug behavior isolated	SIT1	5
I68	List of issues/decisions considered during design	SYS12	1
I24	List of executed statements and procedure calls, variable values	SYS8	7
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	3
I66	Expected program state, e.g. expected variable values when procedure is called	SYS11	4
I74	Good description of the bug and why and how other fixes were done	SIT1	2
I12	Environment, global, local scope	SYS3	2
I78	Dynamic function return values		5
I37	Language definitions, e.g. reserved words, instruction defs, for C, Pascal, etc.	SYS8, SYS18	1
I55	Domain Simulation	SIT2	1
I76	Available tools to aid in understanding		3
I65	Assembly Language Code Segment number (Machine Code)	SYS2	1
I77	Data-flow trace		1

### C.3 Enhancement

There are two enhancement programming sessions, **EN1** and **EN2**.

#### C.3.1 EN1: Domain/Language Expert – Add Functionality

**EN1** is an expert in the operating systems application domain. He is also an expert with the particular operating system he was working on with approximately 3 1/2 years experience. He is very familiar with the programming language used to implement the operating system.

The task he worked on during the programming session involved adding new functionality to an existing operating systems command. The enhancement was embedded in

code that was not written by EN1. At the start of the session, he had already written the enhancement code and was in the process of removing bugs. Throughout the session he would encounter a bug, track it down, fix it and compile it, which usually uncovered another bug. However, within the two hours he was able to remove all the bugs and watch the enhancement work correctly using local tests. At the end of session he was confident enough in the soundness of the new code to begin regression testing.

Table C.35: EN1: Action-Type Frequencies – Enhancement

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	1	3%
OP2	Determine next prgm. seg to examine	9	25%
OP3	Generate/revise hypothesis re: functionality	1	3%
OP7	Investigate oversight	2	6%
OP8	Failed hypothesis	4	11%
OPKNOW	Top-down knowledge	19	52%
<i>Total</i>	<i>Top-Down Model Actions</i>	36	12%
SIT1	Gain situation knowledge	3	5%
SIT2	Develop questions	2	3%
SIT3	Determine answers to questions	1	2%
SIT4	Chunk & store	7	11%
SIT5	Determine relevance of sit. know.	1	2%
SIT7	Generate hypothesis	4	7%
SIT8	Determine understand strategy	1	2%
SIT9	Determine if error exists (missing funct)	2	3%
SIT10	Failed hypothesis	1	2%
SIT11	Mental simulation	2	3%
SITCONF	Confirmed hypothesis	1	2%
SITKNOW	Situation model knowledge	35	58%
<i>Total</i>	<i>Situation Model Actions</i>	60	20%
SYS1	Read intro code comments/related docs	2	21%
SYS2	Determine next prg segmt to examine	6	3%
SYS3	Examine next module in sequence	14	7%
SYS4	Examine next module in cntrl-flow	11	6%
SYS5	Examine data structs & definitions	2	1%
SYS7	Chunk & store knowledge	17	8%
SYS8	Generate hypothesis	30	15%
SYS10	Determine understand strategy	9	5%
SYS11	Generate new task	14	7%
SYS12	Generate question	8	4%
SYS13	Determine if looking at right code	2	1%
SYS14	Change direction	2	1%
SYS15	Generate/consider different code changes	3	2%
SYS16	Answer question	1	0%
SYS17	Add/Alter code	15	8%
SYS18	Determine location to set breakpt	11	6%
SYS19	Failed hypothesis	7	3%
SYS20	Determine error/omitted code to add	7	3%
SYS21	Mental simulation	7	3%
SYSCONF	Confirmed hypothesis	1	0%
SYSKNOW	Program model knowledge	32	16%
<i>Total</i>	<i>Program Down Model Actions</i>	201	68%

Table C.36: EN1: References and Action Switches Between Models

Number of References	Model	Model Switches - Understand Program		
		Top-Down Model	Situation Model	Program Model
36	Top-Down Model	N/A	7	19
60	Situation Model	11	N/A	18
201	Program Model	15	22	N/A

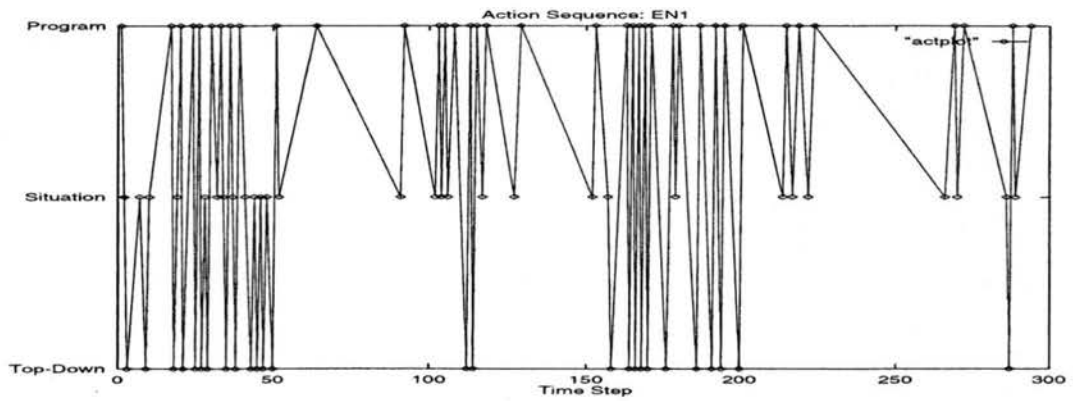


Figure C.24: EN1: Enhancement – Action Sequence

Table C.37: EN1: Hypothesis-Type Frequencies – Enhancement

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH9	Permissions/Environment set correctly/ Tool functionality	1	50%
	OPH16	Level & structure of code/scope	1	50%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	2	6%
Situation Model	SITH2	Function/code block execution order/state	1	25%
	SITH3	Function/procedure function, call function	1	25%
	SITH4	Effect of running program	1	25%
	SITH7	Existence of functionality/algorithm/ variable	1	25%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	4	11%
Program Model	SYSH2	Function/procedure function	1	3%
	SYSH5	Location/type/existence of function call	2	7%
	SYSH6	Statement execution order/state	3	10%
	SYSH7	Variable value/defaults	7	24%
	SYSH8	(Non-)Existence of construct (var/code)	1	3%
	SYSH16	Code correctness, cause/location of error	15	50%
	SYSH18	Location to add code/alternatives	1	3%
	<i>Total</i>	<i>Program Model Hypotheses</i>	30	83%

Table C.38: EN1: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
2	Top-Down Model	N/A	0	4
4	Situation Model	0	N/A	4
30	Program Model	5	3	N/A

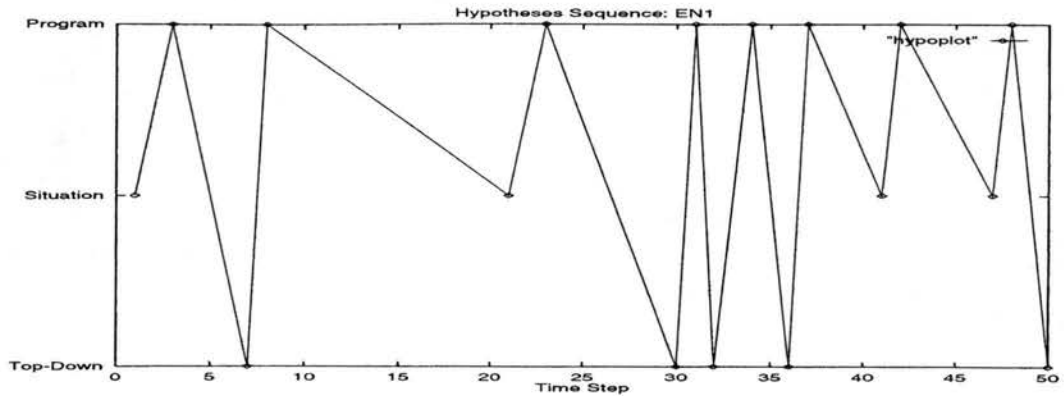


Figure C.25: EN1: Enhancement – Hypotheses Sequence

Table C.39: EN1: Enhancement - Information Needs

Code	Information Need	Action Codes	Subject Frequencies
I7	Domain concept descriptions	OP1	1
I61	Connected domain-program-situation model knowledge	SYS7	4
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	7
I9	List of browsed locations	OP2, OP4, SYS2, OP4	6
I3	Highlighted begin/ends of control blocks	SYS3, SYS7, SYS8, SYS10, SYS24	1
I22	History of past modifications	SYS3	1
I73	Bug behavior isolated	SIT1	5
I24	List of executed statements and procedure calls, variable values	SYS8	2
I17	Location of desired code segment	SYS1, SYS2	2
I67	Location of where to put changes	SYS11	5
I36	Sequence of locations where ID is used	OP2	1
I70	State of system when crashed	SYS18	2
I48	Code formatted in expected way	SYS3	1
I25	Exact location to set breakpoint	SYS18	3
I21	Organized functions into categories in which functions are related	OP2	1
I65	Assembly Language Code Segment number (Machine Code)	SYS2	2
I69	Possible errors and severities, case that always need safety checks	SYS17	1



### C.3.2 EN2: Language/Domain Expert Add Function

EN2 worked on an enhancement task that involved adding a check for disk-full to a schematic capture CAD system. He was an expert in the domain, having worked on the system for several years. The subsystem that handles file creation was the general area where the new code would be added. He was not very familiar with this part of the system. Prior to the programming session he had used a debugger to step through the code so he could get an idea of the structure. The system was written in MainSail, which is a structured programming language similar to Pascal. EN2 had eight years of experience with MainSail.

The main task EN2 worked on during the programming session was to determine the correct location to add the check for disk-full. The strategy he took was to load the code into the debugger and set breakpoints at strategic locations where he thought the enhancement might be added. He successively set and cleared breakpoints so he could look at details for relevant sections and skip details for those sections he judged unimportant to the task. By the end of the session, he had located where the enhancement should go and had written pseudo-code that would eventually be replaced by real code.

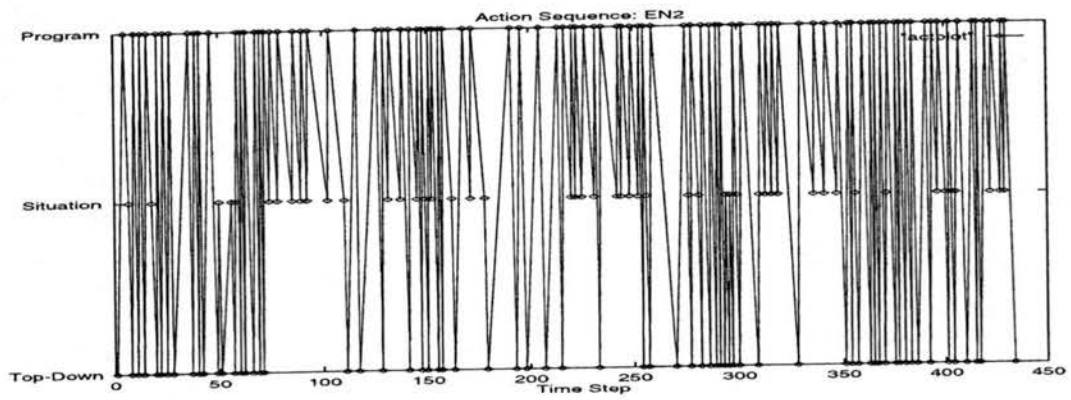


Figure C.26: EN2: Enhancement – Action Sequence

Table C.40: EN2: References and Action Switches Between Models

Number of References	Model	Model Switches - Understand Program		
		Top-Down Model	Situation Model	Program Model
156	Top-Down Model	N/A	8	58
53	Situation Model	17	N/A	33
226	Program Model	49	42	N/A

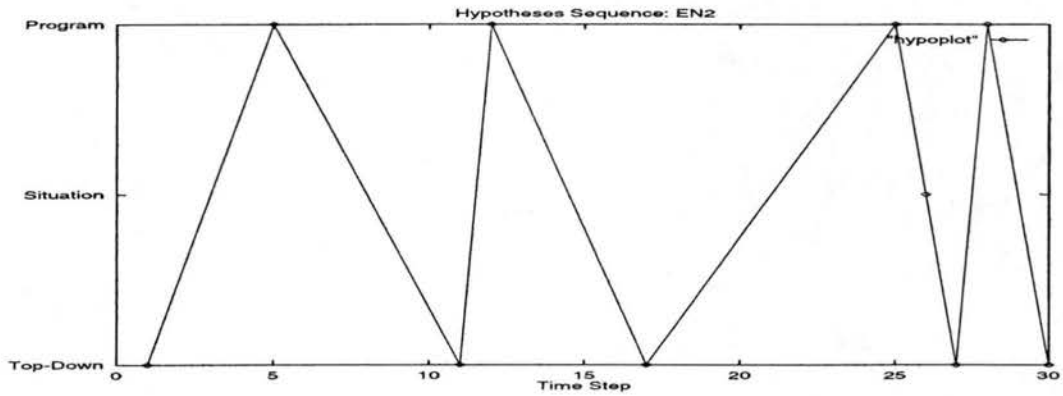


Figure C.27: EN2: Hypotheses Sequence

Table C.41: EN2: Action-Type Frequencies – Enhancement

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	1	1%
OP2	Determine next prgm. seg to examine	6	4%
OP3	Generate/revise hypothesis re: functionality	13	8%
OP4	Determine relevance of prgm segment	7	4%
OP5	Determine if prgm seg needs detail understand	1	1%
OP6	Determine understand strategy	5	3%
OP8	Failed hypothesis	1	1%
OP9	Mental simulation	1	1%
OP11	High-level change plan/alternatives	17	11%
OP13	Study/initiate program execution	10	6%
OP14	Compare program segments	3	2%
OP15	Generate questions	7	4%
OP16	Answer questions	4	3%
OP17	Chunk & store knowledge	13	8%
OP20	Generate task	35	22%
OPCONF	Confirmed hypothesis	1	1%
OPKNOW	Top-down knowledge	31	20%
<i>Total</i>	<i>Top-Down Model Actions</i>	156	36%
SIT1	Gain situation knowledge	1	2%
SIT2	Develop questions	1	2%
SIT4	Chunk & store	31	58%
SIT6	Determine next info to gain	1	2%
SIT7	Generate hypothesis	1	2%
SIT11	Mental simulation	11	21%
SITKNOW	Situation model knowledge	7	13%
<i>Total</i>	<i>Situation Model Actions</i>	53	12%
SYS1	Read intro code comments/related docs	13	6%
SYS2	Determine next prg segmt to examine	10	4%
SYS3	Examine next module in sequence	11	5%
SYS4	Examine next module in cntrl-flow	17	8%
SYS5	Examine data structs & definitions	1	0%
SYS7	Chunk & store knowledge	40	18%
SYS8	Generate hypothesis	11	5%
SYS10	Determine understand strategy	3	1%
SYS11	Generate new task	24	11%
SYS12	Generate question	7	3%
SYS13	Determine if looking at right code	3	1%
SYS15	Generate/consider different code changes	25	11%
SYS16	Answer question	1	0%
SYS17	Add/Alter code	10	4%
SYS18	Determine location to set breakpt	6	3%
SYS19	Failed hypothesis	1	0%
SYS20	Determine error/omitted code to add	17	8%
SYS21	Mental simulation	13	6%
SYS23	Search for var defines/use	1	0%
SYSCONF	Confirmed hypothesis	2	2%
SYSKNOW	Program model knowledge	10	4%
<i>Total</i>	<i>Program Down Model Actions</i>	226	52%

Table C.42: EN2: Hypothesis-Type Frequencies Enhancement

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	9	56%
	OPH5	Existence of installed (running) program	2	12%
	OPH10	Location to add functionality	3	20%
	OPH11	Comparison of functionality at high level	2	12%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	16	57%
Situation Model	SITH3	Function/procedure function, call function	1	100%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	1	4%
Program Model	SYSH2	Function/procedure function	3	27%
	SYSH5	Location/type/existence of function call	1	9%
	SYSH7	Variable value/defaults	1	9%
	SYSH13	Code block/procedure comparison	1	9%
	SYSH16	Code correctness, cause/location of error	1	9%
	SYSH18	Location to add code/alternatives	4	37%
	<i>Total</i>	<i>Program Model Hypotheses</i>	11	39%

Table C.43: EN2: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches - Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
16	Top-Down Model	N/A	0	4
1	Situation Model	1	N/A	0
11	Program Model	3	1	N/A

Table C.44: EN2: Enhancement - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	4
I61	Connected domain-program-situation model knowledge	SYS7	3
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	2
I9	List of browsed locations	OP2, OP4, SYS2 OP4	5
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	6
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	3
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	1
I22	History of past modifications	SYS3	1
I14	Call Graph Display	OP1, OP2, SYS3	1
I68	List of issues/decisions considered during design	SYS12	4
I24	List of executed statements and procedure calls, variable values	SYS8	1
I42	Utility functions definitions and comments explaining why it was rewritten	OP2	1
I17	Location of desired code segment	SYS1, SYS2	3
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	1
I67	Location of where to put changes	SYS11	4
I79	Ripple Effect – Procedure affected by change, include port affects, dependencies in in Make file		1
I62	Predefined (constant) variables and values	SYS8	3
I13	Conditions under which a branch is taken or not. Include variable values	SYS3	1
I30	Where variable is toggled, when and why, where passed to and why	SYS2, SYS12	2
I41	Call graph with extraneous information not relevant elided	OP2	1
I25	Exact location to set breakpoint	SYS18	1
I46	What kind of architecture dependent code currently exists and is functional what architecture hooks exist so code can be easily added.	OP3	1
I77	Data-flow trace		1

#### C.4 Adaptation

There are two adaptive maintenance programming sessions, AD1 and AD2.

### C.4.1 AD1: Language/Domain Expert Port Program

The task AD1 worked on was unlike any other we have observed. It involved porting whole programs from one machine platform to another and making sure they ran properly. The observed task was one part of a larger system porting project.

The programs run under an X window environment and were designed for demonstrating graphics capabilities. Porting requires copying the programs to the target machine, recompiling the program, and running it to make sure it operates properly. AD1 had expertise in several different areas including the source operating system, X-window programming, and operating systems in general. His language expertise includes C, shell scripts, and make files. He had only 6 months of experience with the target operating system. At the beginning of the programming session, AD1 had already copied, compiled, and run one of the programs. He had also determined that the program was not functioning properly. Therefore, the session started with a bug fix. He quickly fixed the problem and went on to successfully port several other X programs within two hours.

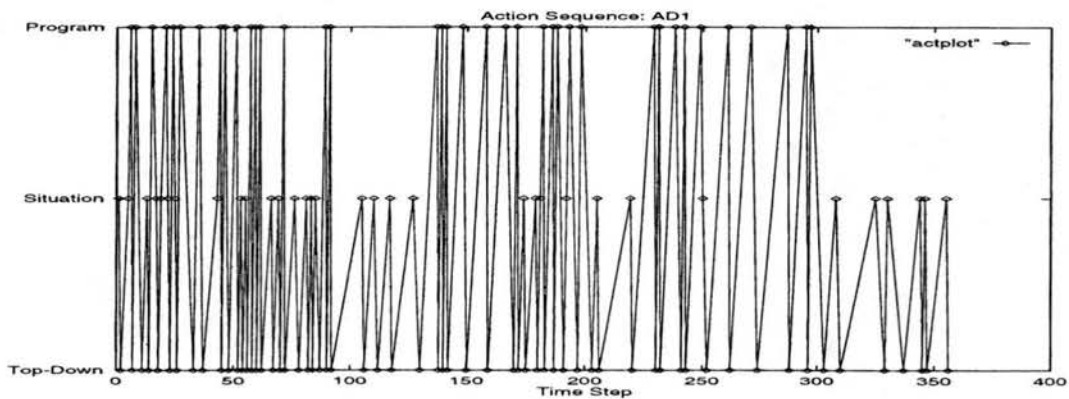


Figure C.28: AD1: Port Programs - Action Sequence

Table C.45: AD1: Action-Type Frequencies – Port Programs

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	8	3%
OP2	Determine next prgm. seg to examine	3	1%
OP3	Generate/revise hypothesis re: functionality	21	9%
OP4	Determine relevance of prgrm segment	3	1%
OP6	Determine understand strategy	11	5%
OP8	Failed hypothesis	4	2%
OP9	Mental simulation	3	1%
OP11	High-level change plan/alternatives	9	4%
OP12	Observe buggy behavior	9	4%
OP13	Study/initiate program execution	34	14%
OP14	Compare program segments	3	1%
OP15	Generate questions	7	3%
OP16	Answer questions	2	1%
OP17	Chunk & store knowledge	10	4%
OP18	Change directions	8	3%
OP20	Generate task	30	13%
OP22	Examine execution results	14	6%
OPCONF	Confirmed hypothesis	9	4%
OPKNOW	Top-down knowledge	51	21%
<b>Total</b>	<b>Top-Down Model Actions</b>	<b>239</b>	<b>67%</b>
SIT1	Gain situation knowledge	5	10%
SIT2	Develop questions	1	2%
SIT3	Determine answers to questions	1	2%
SIT4	Chunk & store	3	6%
SIT7	Generate hypothesis	14	28%
SIT9	Determine if error exists (missing funct)	1	2%
SIT10	Failed hypothesis	4	8%
SITCONF	Confirmed hypothesis	3	6%
SITKNOW	Situation model knowledge	18	36%
<b>Total</b>	<b>Situation Model Actions</b>	<b>50</b>	<b>14%</b>
SYS1	Read intro code comments/related docs	2	3%
SYS2	Determine next prg segmt to examine	4	6%
SYS7	Chunk & store knowledge	2	3%
SYS8	Generate hypothesis	3	4%
SYS10	Determine understand strategy	2	3%
SYS11	Generate new task	2	3%
SYS12	Generate question	6	8%
SYS15	Generate/consider different code changes	4	6%
SYS16	Answer question	6	8%
SYS17	Add/Alter code	10	14%
SYS20	Determine error/omitted code to add	2	3%
SYS21	Mental simulation	4	6%
SYS22	Compare code versions	2	3%
SYSCONF	Confirmed hypothesis	2	3%
SYSKNOW	Program model knowledge	19	27%
<b>Total</b>	<b>Program Down Model Actions</b>	<b>70</b>	<b>19%</b>

Table C.46: AD1: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
239	Top-Down Model	N/A	28	33
50	Situation Model	28	N/A	5
70	Program Model	34	4	N/A



Table C.47: AD1: Hypothesis-Type Frequencies – Port Programs

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH4	I/O behavior	1	4%
	OPH5	Existence of installed (running) program	5	22%
	OPH6	Existence of specific functionality	4	17%
	OPH7	Number/type/existence/location of libraries	2	9%
	OPH8	Program functions correctly	4	17%
	OPH9	Permissions/Environment set correctly/ Tool functionality	2	9%
	OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	1	4%
	OPH13	Number/type/location of file	2	9%
	OPH14	Available functionality	1	4%
	OPH15	Approaches/relative difficulty in making change	1	5%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	23	58%
Situation Model	SITH4	Effect of running program	4	29%
	SITH5	Cause of buggy behavior	2	14%
	SITH7	Existence of functionality/algorithm/ variable	1	7%
	SITH8	Program function	7	50%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	14	35%
Program	SYSH15	Relevance of error in Makefile	1	33%
Model	SYSH16	Code correctness, cause/location of error	1	33%
	SYSH17	Changes made correctly	1	34%
	<i>Total</i>	<i>Program Model Hypotheses</i>	3	7%

Table C.48: AD1: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
23	Top-Down Model	N/A	10	1
14	Situation Model	11	N/A	1
3	Program Model	1	1	N/A

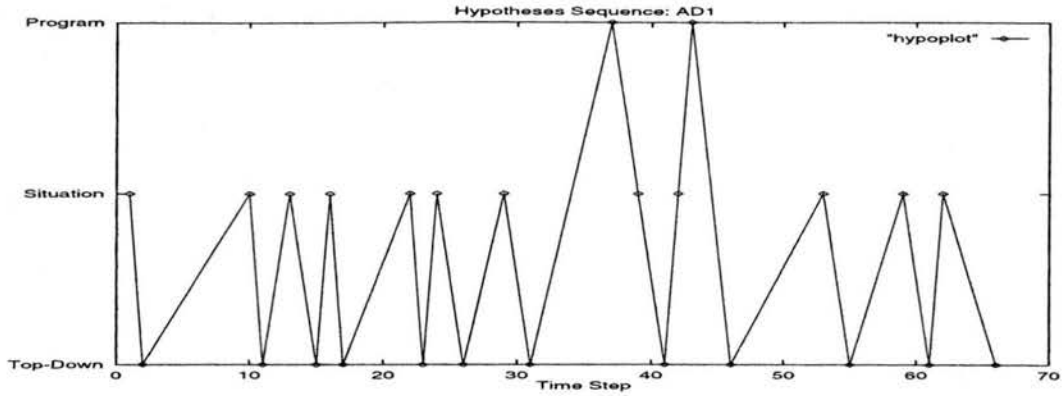


Figure C.29: AD1:Port Programs – Hypotheses Sequence

Table C.49: AD1: Port Programs - Information Needs

Code	Information Need	Action Codes	Subject Frequencies
I7	Domain concept descriptions	OP1	11
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	1
I9	List of browsed locations	OP2, OP4, SYS2 OP4	1
I22	History of past modifications	SYS3	1
I73	Bug behavior isolated	SIT1	1
I24	List of executed statements and procedure calls, variable values	SYS8	2
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	4
I17	Location of desired code segment	SYS1, SYS2	3
I6	Location and description of library routines and system calls	OP1, OP2, OP20, SYS11	1
I74	Good description of the bug and why and how other fixes were done	SIT1	2
I49	File name of current file	SYS3	1
I55	Domain Simulation	SIT2	1
I71	Concise error description	SYS20	2
I40	List of capabilities that may need to be provided	OP1	2

## Opportunistic Dynamic Resolution Process

We analyzed relationships between hypotheses (for subject **AD1**), how sequences of goals and hypotheses affect achieving the overall goal of the task, and tried to identify an overall strategy of hypothesis generation and resolution.

The protocol contained 20 major goals. For each goal, the analysis excerpted associated hypotheses, supporting comprehension actions, and any subsidiary goals, hypotheses, and actions. The result of the analysis is represented graphically. Each figure shows analysis results for an individual top-level goal. Ellipses represent goals, rectangles are hypotheses, and actions are triangles. Arrows show the flow of goal/hypothesis resolution. The flow of goals, hypotheses, and actions over time is determined by a left to right depth-first traversal of the tree. Each figure is accompanied by descriptions of each goal and supporting subgoals, hypotheses, and actions.

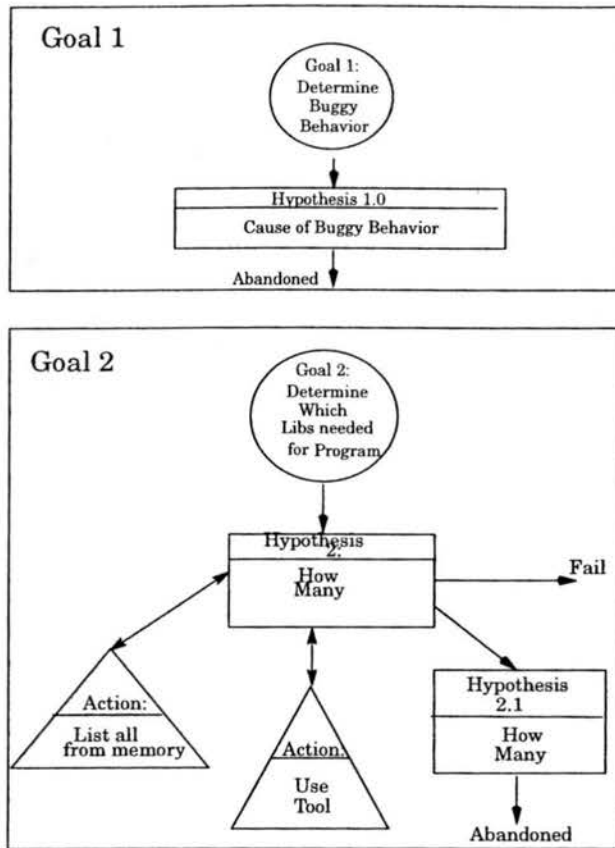


Figure C.30: AD1: Goals 1 and 2

GOAL 1

What he was doing: Invoked the program to watch it run incorrectly.

Goal 1: Determine what was wrong by watching program behavior.

**Hypothesis 1:** Cause of the buggy behavior and generation of alternative hypotheses.

This hypothesis was never returned to and is considered abandoned.

GOAL 2

What he was doing: Looked at an Imake file.

Goal 2: To determine what libraries are needed by the program.

**Hypothesis 2:** He guessed at how many; then listed them by name from memory. He tried to confirm this by using a tool that listed the libraries, including libraries used by other libraries.

*Sub-Hypothesis 2.1:* Result of tool output led to acknowledging his original hypothesis

(#2) was incorrect; there were more than his original estimate. He hypothesized that there were at least 4 libraries involved, but immediately abandoned this hypothesis because of the amount of work it would take to confirm it. He judged the answer to be irrelevant to the overall goal of porting the program.

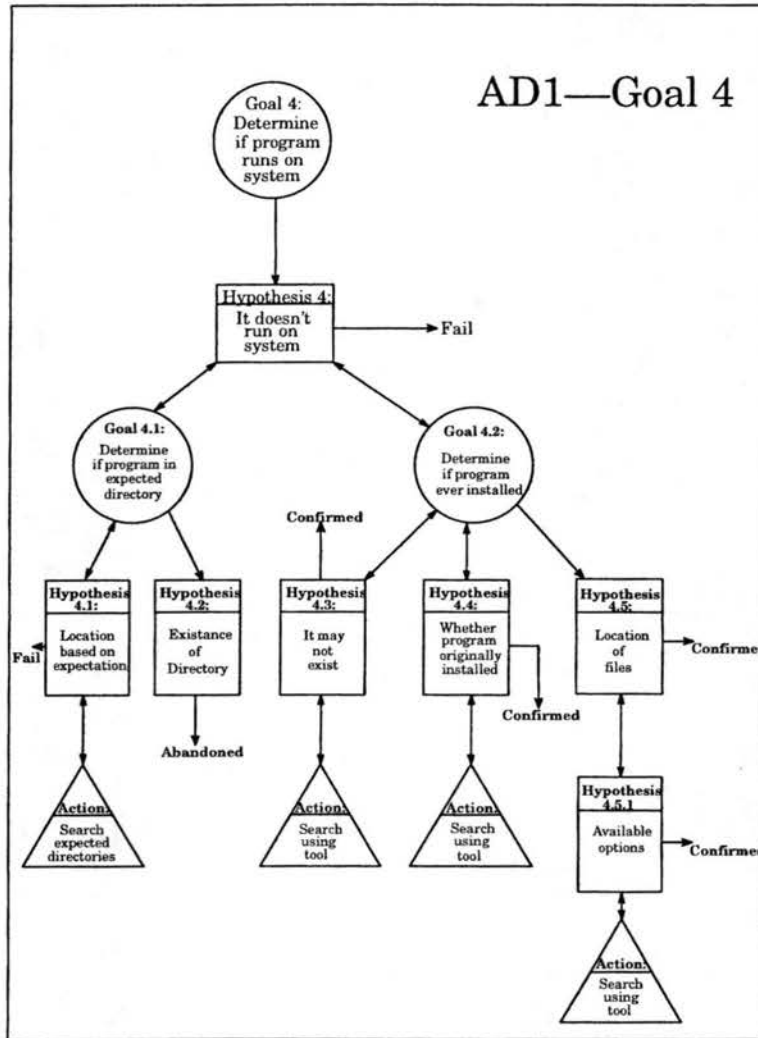


Figure C.31: AD1: Goal 4

#### GOAL 4

What he was doing: See if the program runs on a particular system. He had never seen it on that platform.

Goal 4: Get it to run on his system if it was there.

**Hypothesis 4:** Doesn't exist on that system. In order to determine if it was there or not he generated two sub-goals.

*Sub-Goal 4.1:* Determine if the program is where he expected.

**Sub-Hypothesis 4.1:** He guessed at the location. It failed after a search of the directory he expected it to be in. This led directly to the next sub-hypothesis.

**Sub-Hypothesis 4.2:** Hypothesis about the existence of another directory. Hypothesis was abandoned in favor of alternative goal/hypothesis sequence #4.2 instead of searching for the directory itself. The alternative directory was never mentioned again.

*Sub-Goal 4.2:* Determine if the program was ever installed as part of the operating system.

**Sub-Hypothesis 4.3:** Program doesn't exist on his system (space limitations). He invoked a system-wide *find* command from a particular place within the file system and then went off to other things while the find was executing. Eventually, the *find* returned with nothing found. Thus, his hypothesis was confirmed; the program did not exist on that part of his system.

**Sub-Hypothesis 4.4:** He then decided that he did not use the *find* command from the correct place. He reasoned that if he went to the root directory on his system and issued a find from there, he would be able to tell if the program had ever been installed. Again, he issued the *find* command and went on with other things. The *find* command returned with nothing found, confirming his hypothesis that the program was never installed.

**Sub-Hypothesis 4.5:** At this point he knew the program was never installed but suspects the program is in the release of the operating system somewhere. He hypothesized it was somewhere in the OS release bits. This led to another sub-hypothesis about the correct *find* command to issue.

*Sub-Hypothesis 4.5.1:* He thinks there are options to tell the *find* command program to search case insensitive. He looks for the option by reading man pages on the computer. This hypothesis is confirmed when he finds the correct option.

He then uses the find command with the appropriate options from within the O/S direc-

tory. The tool finds the files. His top-level hypothesis (#4) that the program does not exist, fails.

# AD1—Goal 7

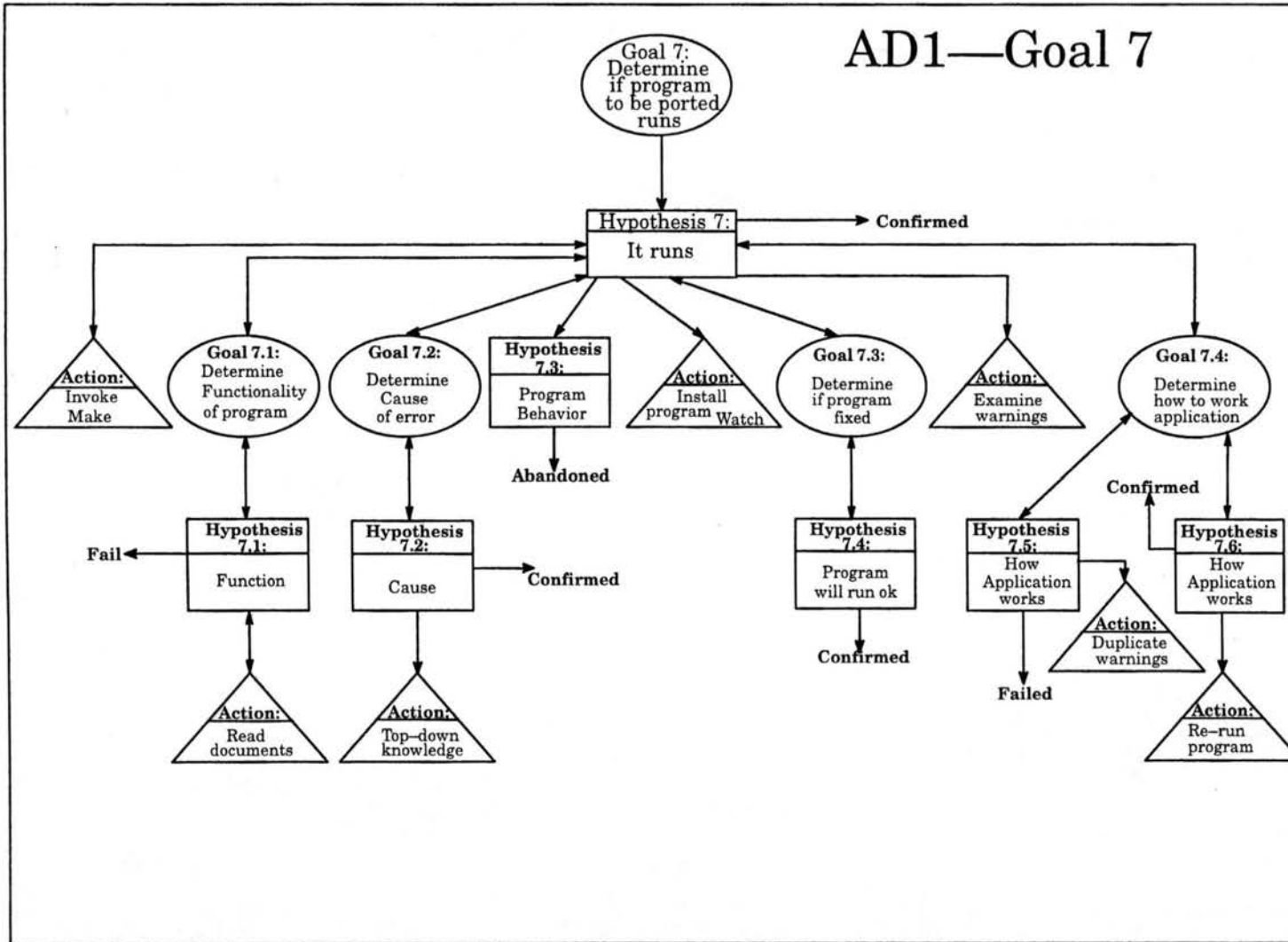


Figure C.32: AD1: Goal 7



## GOAL 7

What he was doing: Try to port another client.

Goal 7: Determine if the program runs.

**Hypothesis 7:** He hypothesized that the program runs because it had successfully compiled at some point prior to the programming session. He reasoned that he could re-build the program by executing a *Make* command and then invoke the newly built client to watch it run. Several subgoals, hypotheses, and actions followed that confirmed the program was running as expected.

**Goal 7.1:** Determine the program's functionality.

**Hypothesis 7.1:** He hypothesized that this program was a ping-pong game. He brought up the associated man pages and determined that instead it was a physical simulation of an ideal gas in a heated box.

**Goal 7.2:** He noticed the program was not behaving as explained in the *man* pages. He decided to determine the cause.

**Hypothesis 7.2:** Hypothesized the cause as related to warnings about missing files. He used top-down knowledge about client architecture to confirm this hypothesis.

**Hypothesis 7.3:** He looked at the *Make* file and hypothesized that the missing files would be overwritten during the build process. This hypothesis was never returned to and considered abandoned.

**Action:** He installed the program and then invoked it so he could watch it run. During the install process, he returned to Goal 4 before coming back to this one.

**Goal 7.3:** After installing the program he wanted to confirm that it could find the missing files by executing the program.

**Hypothesis 7.4:** Hypothesized that the program will run this time, that the missing files will be found. This hypothesis was immediately confirmed (no warning messages).

**Action:** The program was run, new warning messages were examined.

**Goal 7.4:** To understand the new warning messages he decided to determine how to use the application.

**Hypothesis 7.5:** Hypothesized about how the application works. He was convinced he had to use a trick to get the program to function correctly. He tried to duplicate the warnings by re-running the application but this time did not get the warnings. This hypothesis failed when he realized he was missing some required input.

**Hypothesis 7.6:** Based on the results of the previous hypothesis, he hypothesized that the program worked differently. He tested his theory and this time the program behaved as expected.

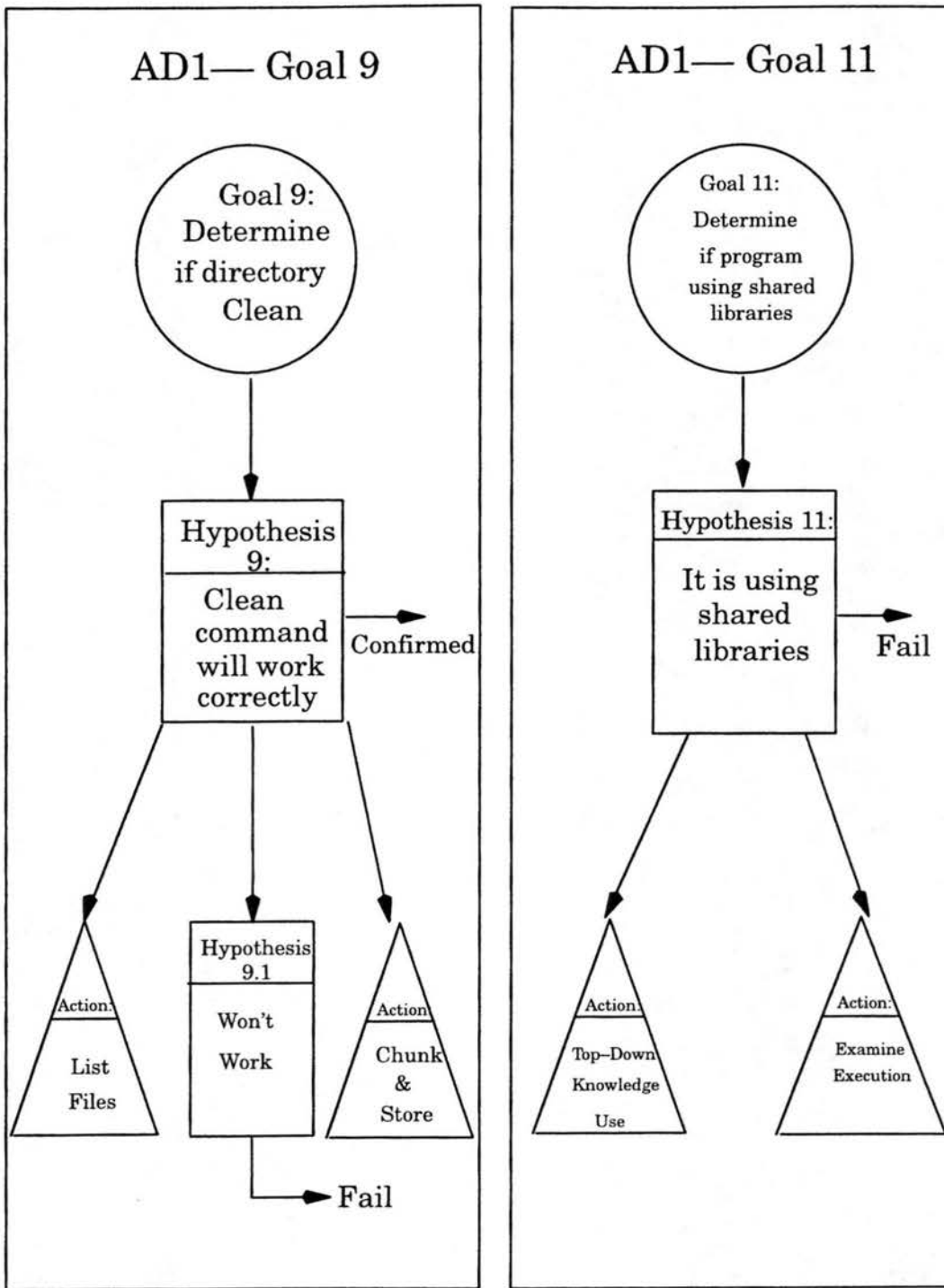


Figure C.33: AD1: Goals 9 & 11

### **Goal 9**

What he was doing: Trying to clean up his directory in order to rebuild the system.

Goal 9: Determine if the directory was clean.

- Hypothesis 9: He ran the make clean command and hypothesized that it would clean up the directory as expected and get rid of some specific files. Once the clean command was complete he listed the files to check if his hypothesis was correct. This hypothesis was confirmed after a brief period of confusion in which an additional hypothesis was generated.
  - Hypothesis 9.1: He looked at the list of files remaining in the directory and was momentarily confused. A file existed that he did not expect to still be in the directory. He hypothesized that the clean did not get rid of all the files he expected it to. This new hypothesis failed when he remembered that he had created the file to store old files into and that the clean command would not erase this file.

### **Goal 11**

What he was doing: Trying to build an executable version of a client program.

Goal 11: To determine if the program building process was linking in shared libraries.

- Hypothesis 11: As he watched the build process he hypothesized that the build was going to fail because it was trying to build the program with shared libraries. He reasoned, using top-down knowledge, that the configuration file requires the build to use shared libraries. After watching output from the build process, he determined that the build could not be using the shared libraries because the process was taking too long.

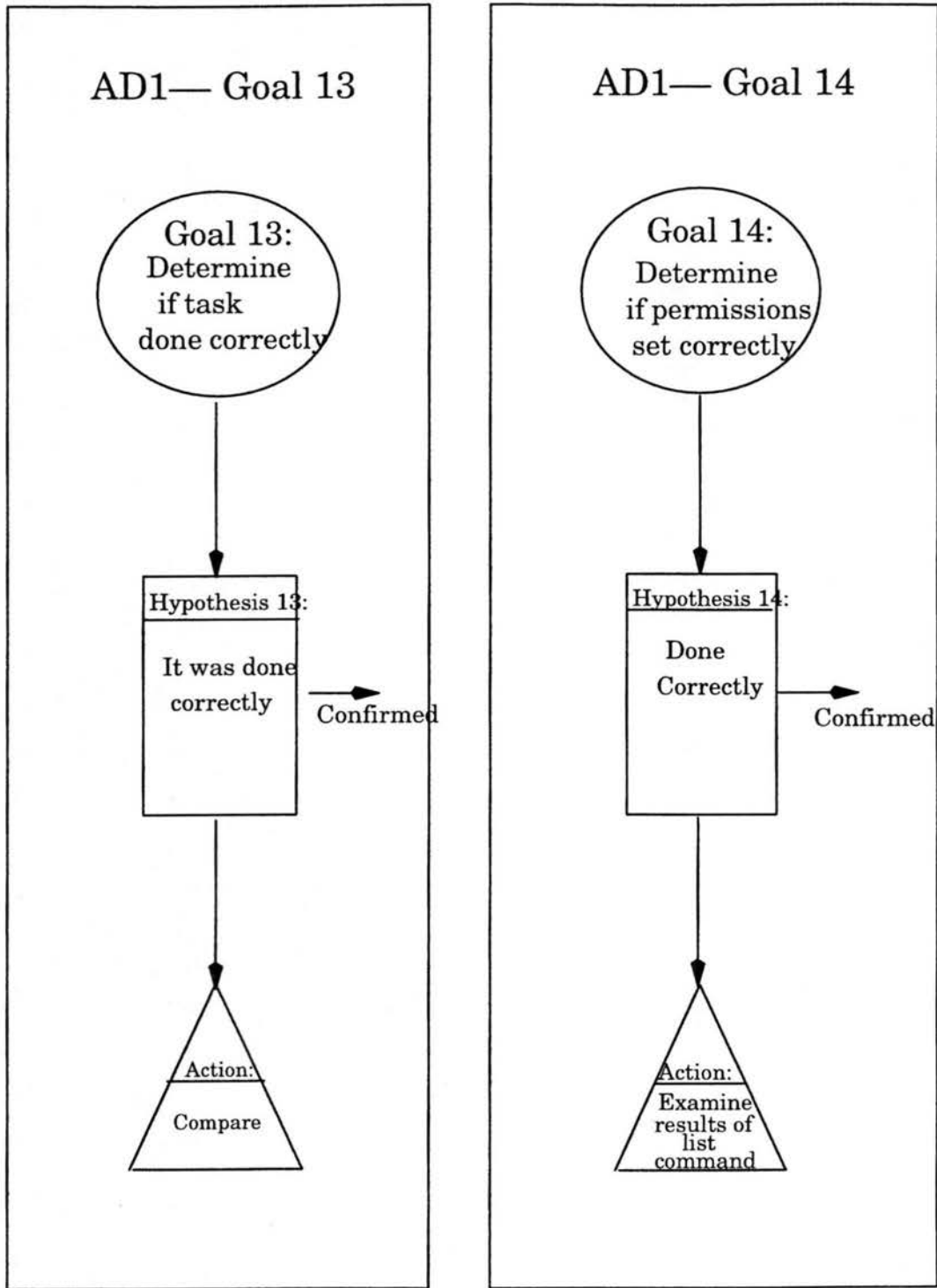


Figure C.34: AD1: Goals 13 & 14

### **Goal 13**

What he was doing: He had made a change to a make file and wanted to verify what he had done.

Goal 13: To determine if the changes had been done correctly.

- Hypothesis 13: He hypothesized that he had done the task correctly. To confirm this, he did a file compare between an old version and the version he had just changed. By examining the differences he determined he had done the task correctly.

### **Goal 14**

What he was doing: He had finished making the necessary changes for porting the client, had checked in the new files, and wanted to make sure the permissions were set so that nothing was writable.

Goal 14: To determine that the permissions where set correctly after checking in all the files.

- Hypothesis 14: He assumed that the permissions were set correctly by virtue of the change control system he was using. He wanted to double check this so he listed out all the files and their permission settings. His hypothesis was confirmed when he saw that all the files had read only permissions only.

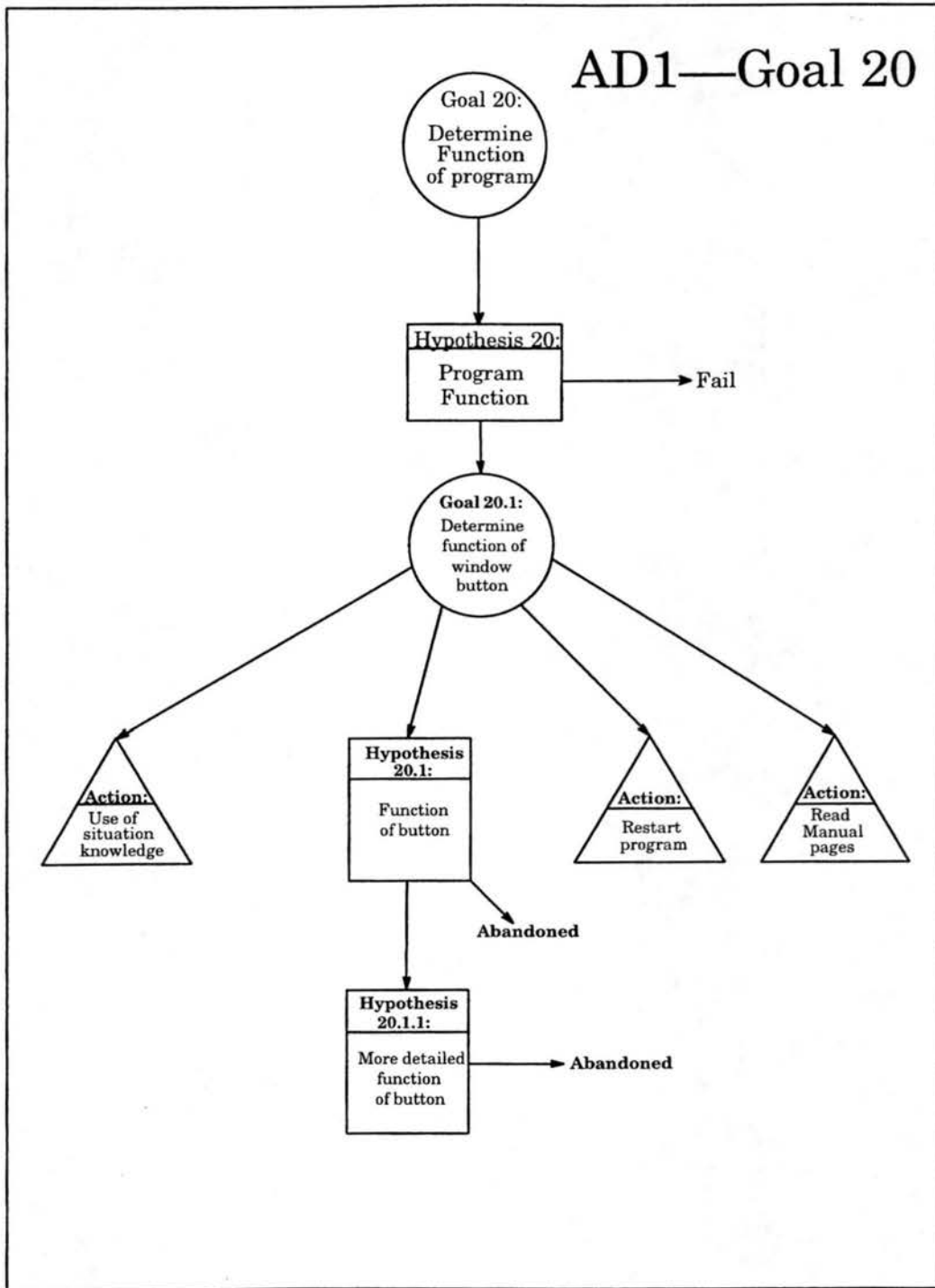


Figure C.35: AD1: Goal 20

## Goal 20

What he was doing: He was working on porting another client he had never seen before. The program was never built prior to this session and he was trying to figure out the functionality of the program.

Goal 20: To Determine the function of the program.

Hypothesis 20: He had successfully built the program and invoked it to watch it run. Examining it, he assumed it was some kind of a graphics editor. The program contained a window with buttons and he immediately generated a goal with subsidiary hypotheses about the function of these buttons. Eventually, this hypothesis failed after reading documentation and finding out it was not a graphics editor.

- Goal 20.1: He saw a button and wanted to determine what the function of the button was. The button was labeled "Record" and using situation knowledge from previous experience he generated a hypothesis about what the record button does.
  - Hypothesis 20.1: When he saw the button label he assumed the function to be a "Save" function. This assumption led to another, more detailed hypothesis about how the save is done. This hypothesis was abandoned and never referred to again during the session.
    - \* Hypothesis 20.1.1: Since the name of the button was "Record", he assumed that the function of the button was to record key strokes and save them in a file. This hypothesis was abandoned and never referred to again during the session.
- Both hypotheses 20.1 and 20.1.1 were abandoned when he ran into a problem caused by the windows environment he was working in and had to kill the program. He then restarted the program and decided at that point to read the on-line man pages. It was after reading these pages that he determined that his original hypothesis (#20) was incorrect. In fact the program did something entirely different than he originally thought.

## Overall Comprehension Process

Figure 5.7 shows the sequence of all 20 goals. Circles represent the goals and arrows indicate sequence. A solid arrow between two goals means the goal was complete before the next goal was begun. A dashed arrow represents interleaving between goals. The figure shows an obvious understanding process. Goals are resolved sequentially unless interleaving of a fairly simple goal can be accomplished efficiently. The subject had experience with the domain and the task. He applied an arsenal of techniques to resolving the goals he generated. If he perceived a hypothesis or goal to be too difficult to resolve, he usually

abandoned it in favor of a more efficient alternative. If the return on investment of resolving a hypothesis was low, the hypothesis or goal was abandoned. If the goal or hypothesis seemed relatively easy to resolve, he usually pursued it as long as it was efficient and as long as it seemed the results would be worth the effort.

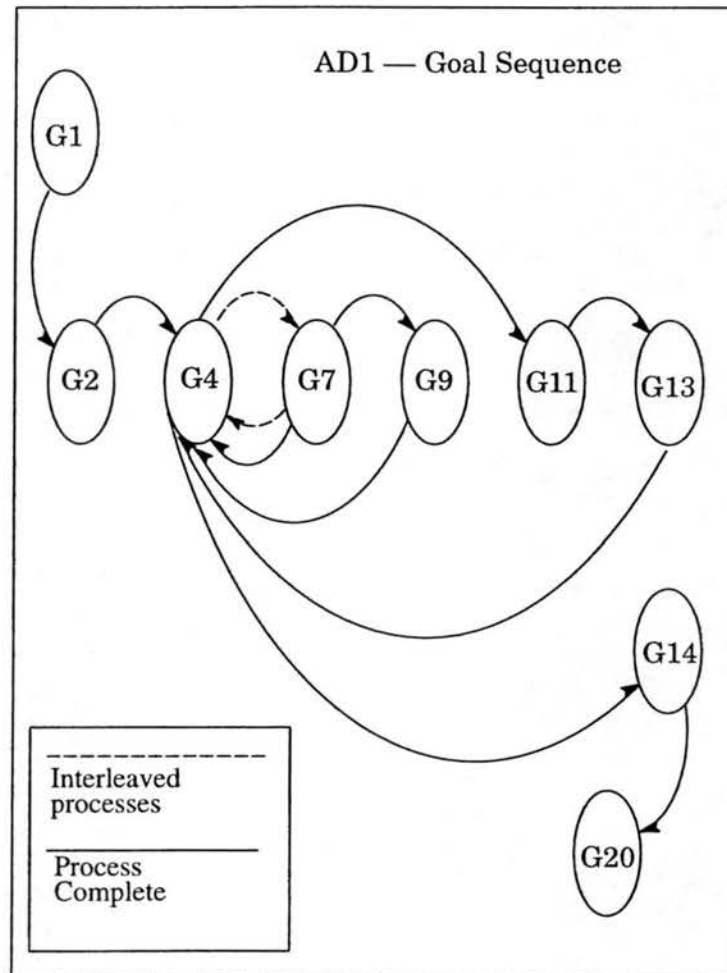


Figure C.36: AD1: Goals Completion



### C.4.2 AD2: Language Novice/Domain Expert Add Function

AD2 worked on a porting task that included moving NFS Disk-less functionality from one Unix platform to another. The task was to create a prototype of the functionality on the new platform for evaluation purposes, i.e., to determine whether this adaptation was worth doing for the new platform. AD2 had seven years of Unix operating systems experience and seven years of C programming language experience. He had worked on the kernel code of the target platform for the previous 6 years.

Prior to the programming session, AD2 had moved code from the source platform to the target platform. He spent approximately one third of the programming session time setting up his environment so he could test whether the newly ported code would work correctly. The strategy he applied was to set up the environment, run a test that would exercise the new functionality on the target system, and track down defects as they occurred. By the end of the programming session he had fixed several problems but he was not completely successful in the porting the program. System memory problems caused a deadlock situation on the target system.

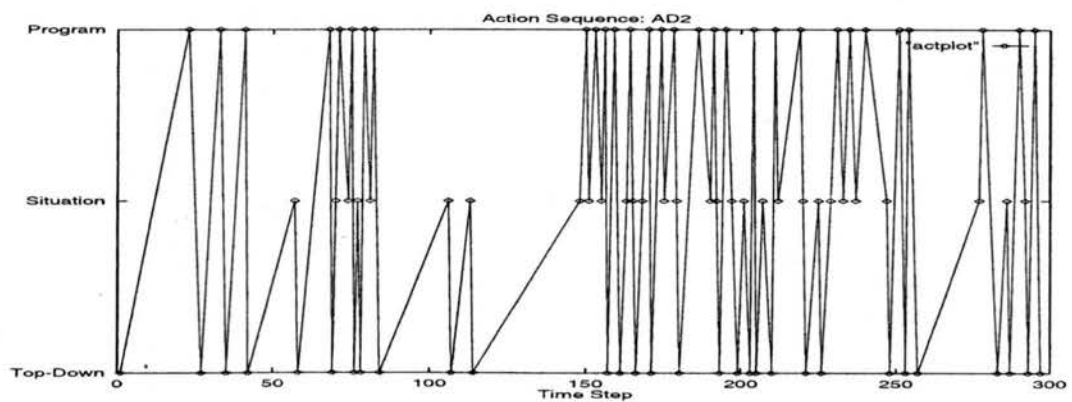


Figure C.37: AD2: Port Program/Add Function – Action Sequence

Table C.50: AD2: Action-Type Frequencies – Port Program/Add Function

<i>Tag</i>	<i>Action-Type</i>	<i>Ttl. Refs</i>	<i>% Ttl.</i>
OP1	Gain high-level Program overview	2	1%
OP2	Determine next prgm. seg to examine	1	1%
OP3	Generate/revise hypothesis re: functionality	41	18%
OP4	Determine relevance of prgrm segment	3	1%
OP6	Determine understand strategy	7	3%
OP7	Investigate oversight	1	1%
OP8	Failed hypothesis	3	1%
OP9	Mental simulation	1	1%
OP11	High-level change plan/alternatives	3	1%
OP12	Observe buggy behavior	6	3%
OP13	Study/initiate program execution	22	10%
OP14	Compare program segments	2	1%
OP15	Generate questions	10	4%
OP16	Answer questions	2	1%
OP17	Chunk & store knowledge	11	5%
OP20	Generate task	55	24%
OP22	Examine execution results	1	0%
OPCONF	Confirmed hypothesis	2	1%
OPKNOW	Top-down knowledge	52	23%
<i>Total</i>	<i>Top-Down Model Actions</i>	225	67%
SIT1	Gain situation knowledge	2	4%
SIT2	Develop questions	3	6%
SIT3	Determine answers to questions	2	4%
SIT4	Chunk & store	12	25%
SIT6	Determine next info to gain	1	2%
SIT7	Generate hypothesis	7	14%
SIT11	Mental simulation	2	4%
SITCONF	Confirmed hypothesis	1	2%
SITKNOW	Situation model knowledge	19	39%
<i>Total</i>	<i>Situation Model Actions</i>	49	15%
SYS1	Read intro code comments/related docs	3	5%
SYS2	Determine next prg segmt to examine	1	2%
SYS3	Examine next module in sequence	6	9%
SYS5	Examine data structs & definitions	2	3%
SYS7	Chunk & store knowledge	13	21%
SYS8	Generate hypothesis	2	3%
SYS10	Determine understand strategy	2	3%
SYS11	Generate new task	9	15%
SYS12	Generate question	1	2%
SYS13	Determine if looking at right code	1	2%
SYS15	Generate/consider different code changes	4	6%
SYS17	Add/Alter code	6	10%
SYS20	Determine error/omitted code to add	1	2%
SYS23	Search for var defines/use	1	2%
SYSCONF	Confirmed hypothesis	1	2%
SYSKNOW	Program model knowledge	8	13%
<i>Total</i>	<i>Program Down Model Actions</i>	61	18%

Table C.51: AD2: References and Action Switches Between Models

<i>Number of References</i>	<i>Model</i>	<i>Model Switches - Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
225	Top-Down Model	N/A	14	15
49	Situation Model	15	N/A	15
61	Program Model	14	16	N/A

Table C.52: AD2: Hypothesis-Type Frequencies – Port Program/Add Function

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	12	29%
	OPH3	Rules of discourse/Expectations	1	2%
	OPH8	Program functions correctly	5	12%
	OPH9	Permissions/Environment set correctly/ Tool functionality	2	5%
	OPH12	How to duplicate warnings/errors; relative difficulty to set-up/test errors	5	12%
	OPH13	Number/type/location of file	1	2%
	OPH14	Available functionality	1	2%
	OPH16	Level & structure of code/scope	1	2%
	OPH18	Location/Status/description/cause of error	14	34%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	42	82%
Situation Model	SITH2	Function/code block execution order/state	1	17%
	SITH5	Cause of buggy behavior	4	66%
	SITH8	Program function	1	17%
		<i>Total</i>	<i>Situation Model Hypotheses</i>	6
Program Model	SYSH1	Variable function	1	33%
	SYSH7	Variable value/defaults	1	33%
	SYSH16	Code correctness, cause/location of error	1	34%
		<i>Total</i>	<i>Program Model Hypotheses</i>	3

Table C.53: AD2: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
42	Top-Down Model	N/A	2	2
6	Situation Model	2	N/A	1
3	Program Model	2	1	N/A

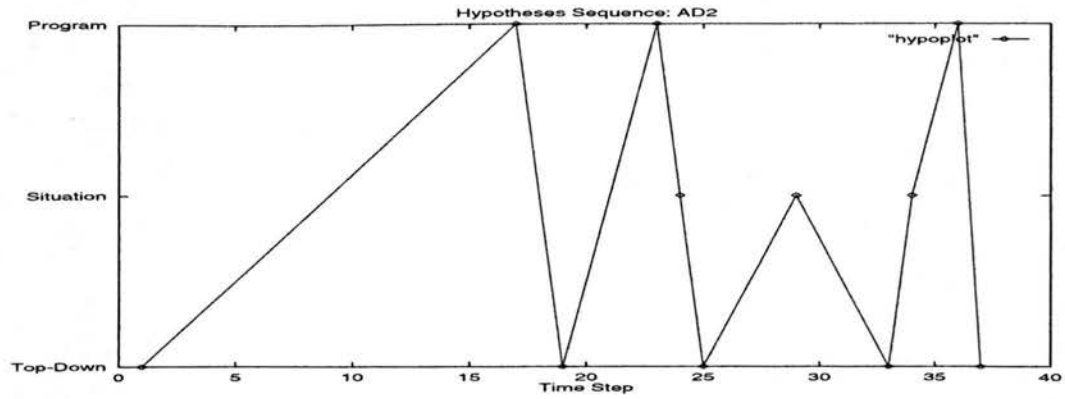


Figure C.38: AD2: Hypotheses Sequence

Table C.54: AD2: Port Program/Add Function - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	2
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	OP2	1
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	1
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	1
I22	History of past modifications	SYS3	5
I24	List of executed statements and procedure calls, variable values	SYS8	2
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	1
I17	Location of desired code segment	SYS1, SYS2	1
I32	If common objects are not used in traditional way, e.g. nil or null	SYS5, SYS8	2
I53	IFDEF'd behavior, conditions under which the branch is taken or not	SIT2	3
I79	Ripple Effect - Procedure affected by change, include port affects, dependencies in in Make file		6
I70	State of system when crashed	SYS18	4
I71	Concise error description	SYS20	1
I76	Available tools to aid in understanding		1
I64	System configuration for rebooting	OP6	3
I80	Program execution sequence, how to use (e.g. man page		1

## C.5 Code Leverage

### C.5.1 L1: Language Expert/Domain Novice – Leverage Program

#### L1: Leverage Small Program – Actions

L1's task was to implement a program in C by leveraging an existing program written in Pascal. The program was a software reliability system that takes defect data and analyzes it according to particular reliability models. L1 had very little experience in reliability models, having taken one graduate level class in the subject. He had at least one and a half years of C programming language and several years of Pascal.

Prior to the programming session he had read a paper on the reliability models implemented in the program and he had run the program and watched the output. At the time of the programming session, he was leery of the code because he knew it to be very buggy. His strategy was to start with the main program and read it line by line. One of the main problems he struggled with during the session was how data was stored and read in by the program. He also tried mapping variable names in the program to concepts he had read about in the reliability paper. By the end of programming session, he had decided that the all he could really leverage were the equation implementations.

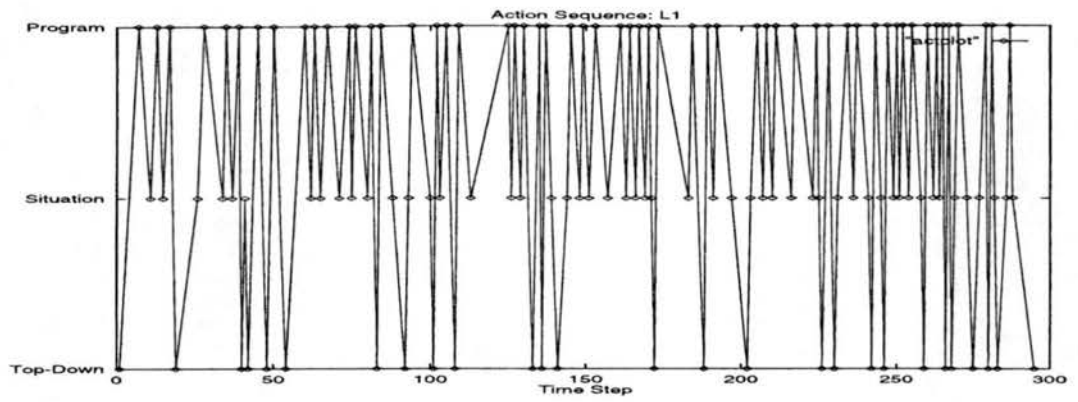


Figure C.39: L1: Leverage Program – Action Sequence

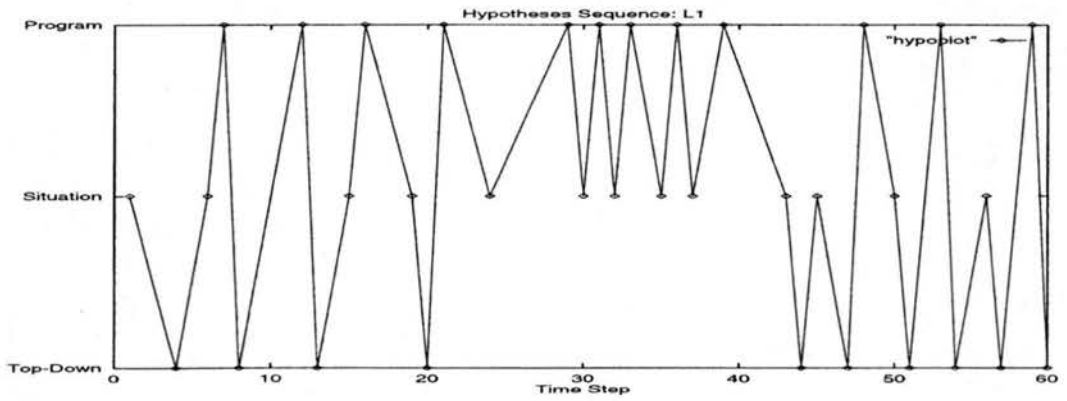


Figure C.40: L1: Leverage Program – Hypotheses Sequence



Table C.55: L1: Action-Type Frequencies – Leverage Program

Tag	Action-Type	Ttl. Refs	% Ttl.
OP1	Gain high-level Program overview	1	2%
OP2	Determine next prgm. seg to examine	1	2%
OP3	Generate/revise hypothesis re: functionality	15	28%
OP6	Determine understand strategy	6	11%
OP8	Failed hypothesis	2	4%
OP9	Mental simulation	1	2%
OP15	Generate questions	2	4%
OP17	Chunk & store knowledge	1	2%
OP20	Generate task	8	15%
OPCONF	Confirmed hypothesis	1	2%
OPKNOW	Top-down knowledge	15	28%
<b>Total</b>	<b>Top-Down Model Actions</b>	<b>53</b>	<b>18%</b>
SIT1	Gain situation knowledge	7	7%
SIT2	Develop questions	1	1%
SIT4	Chunk & store	15	16%
SIT5	Determine relevance of sit. know.	4	4%
SIT6	Determine next info to gain	8	9%
SIT7	Generate hypothesis	20	21%
SIT8	Determine understand strategy	9	10%
SIT10	Failed hypothesis	1	1%
SIT11	Mental simulation	2	2%
SITCONF	Confirmed hypothesis	1	1%
SITKNOW	Situation model knowledge	26	28%
<b>Total</b>	<b>Situation Model Actions</b>	<b>94</b>	<b>32%</b>
SYS1	Read intro code comments/related docs	1	1%
SYS2	Determine next prg segmt to examine	13	9%
SYS3	Examine next module in sequence	20	14%
SYS4	Examine next module in cntrl-flow	1	1%
SYS5	Examine data structs & definitions	13	9%
SYS7	Chunk & store knowledge	28	19%
SYS8	Generate hypothesis	15	10%
SYS10	Determine understand strategy	11	7%
SYS11	Generate new task	16	11%
SYS12	Generate question	1	1%
SYS15	Generate/consider different code changes	1	1%
SYS19	Failed hypothesis	3	2%
SYS21	Mental simulation	4	2%
SYS23	Search for var defines/use	2	1%
SYS24	Search for block begin/end	4	2%
SYSCONF	Confirmed hypothesis	2	1%
SYSKNOW	Program model knowledge	13	9%
<b>Total</b>	<b>Program Down Model Actions</b>	<b>148</b>	<b>50%</b>

Table C.56: L1: References and Action Switches Between Models

Number of References	Model	Model Switches – Understand Program		
		Top-Down Model	Situation Model	Program Model
53	Top-Down Model	N/A	9	17
94	Situation Model	13	N/A	39
148	Program Model	13	43	N/A

Table C.57: L1: Hypothesis-Type Frequencies – Leverage Program

<i>Model</i>	<i>Tag</i>	<i>Hypothesis-Type</i>	<i>Total Refers</i>	<i>Percent of Ttl</i>
Top-Down (Domain) Model	OPH1	Domain Procedure functionality/Concepts	6	40%
	OPH3	Rules of discourse/Expectations	1	7%
	OPH4	I/O behavior	3	20%
	OPH8	Program functions correctly	4	26%
	OPH10	Location to add functionality	1	7%
	<i>Total</i>	<i>Top-Down Model Hypotheses</i>	15	30%
Situation Model	SITH1	Variable function	5	28%
	SITH2	Function/code block execution order/state	4	22%
	SITH3	Function/procedure function, call function	5	28%
	SITH6	Comparison of terms/acronyms /functionality	1	6%
	SITH7	Existence of functionality/algorithm/variable	3	16%
	<i>Total</i>	<i>Situation Model Hypotheses</i>	18	38%
Program Model	SYSH1	Variable function	2	13%
	SYSH2	Function/procedure function	1	6%
	SYSH4	Variable structure	3	19%
	SYSH6	Statement execution order/state	2	13%
	SYSH7	Variable value/defaults	1	6%
	SYSH10	Syntax meaning	1	6%
	SYSH11	Design decisions	1	6%
	SYSH13	Code block/procedure comparison	1	6%
	SYSH14	Code block function	2	13%
	SYSH16	Code correctness, cause/location of error	1	6%
	SYSH17	Changes made correctly	1	6%
	<i>Total</i>	<i>Program Model Hypotheses</i>	16	32%

Table C.58: L1: Hypothesis Generated Switches Between Models

<i>Number of Hypotheses</i>	<i>Model</i>	<i>Model Switches – Understand Program</i>		
		<i>Top-Down Model</i>	<i>Situation Model</i>	<i>Program Model</i>
15	Top-Down Model	N/A	4	5
18	Situation Model	6	N/A	7
16	Program Model	4	8	N/A

Table C.59: L1: Leverage Program - Information Needs

<i>Code</i>	<i>Information Need</i>	<i>Action Codes</i>	<i>Subject Frequencies</i>
I7	Domain concept descriptions	OP1	9
I61	Connected domain-program-situation model knowledge	SYS7	12
I4	Location and Uses of identifiers	OP2, SYS2, SYS3, SYS7, SYS10, SYS11	4
I5	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions.	SYS5, SYS8, SYS11, SYS12, SIT1	5
I2	List of routines that call a specific routine	OP1, OP2, OP4 SYS3, SYS7, SYS9 SYS12	1
I1	Variable definitions including why necessary and how used, default values and expected values	SYS5, SYS8, SYS10 SYS12, SYS17, SYS23	2
I3	Highlighted begin/ends of control blocks	SYS3, SYS7, SYS8, SYS10, SYS24	3
I22	History of past modifications	SYS3	1
I68	List of issues/decisions considered during design	SYS12	2
I27	Directory layout/organization: include files, main file, support files, library files. File structure	OP1, OP2	2
I20	Documentation list and location	OP1	4
I44	List of routines that do most of the domain-type work	OP3	1
I74	Good description of the bug and why and how other fixes were done	SIT1	2
I12	Environment, global, local scope	SYS3	1
I18	I/O parameters, definitions, examples of calls with explanation, what it is used for	SYS3, SYS5, SYS8	5
I37	Language definitions, e.g reserved words, instruction defs, for C, Pascal, etc.	SYS8, SYS18	2
I51	What happens to read in data	SYS5	1