Strategies for designing software

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

• A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/11141](https://dspace.lboro.ac.uk/2134/11141)

Publisher: © Stephen Lang

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
STRATEGIES
FOR
DESIGNING SOFTWARE

By
Stephen Lang

A DOCTORAL THESIS SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE AWARD OF
DOCTOR OF PHILOSOPHY
OF LOUTHBOROUGH UNIVERSITY

April 1998

© Stephen Lang, 1998
Abstract

The purpose of the research described within this thesis is to provide information that can be used to improve the quality of programs and the productivity of the software industry. To fulfil this purpose, the thesis is concerned with the order in which software engineers construct solutions to software problems, the factors that influence the order in which software engineers construct solutions, and the effectiveness of constructing a solution in a particular order. Control strategies describe the order in which solutions are constructed. Four empirical studies investigating control strategies were undertaken to answer the concerns of the thesis.

The aim of the first two studies was to observe the global control strategies employed by software engineers. The two studies collected verbal reports and keystroke behaviour of software engineers using Prolog and C. The results showed that software engineers used a combination of Top-Down control strategies to solve a software problem. A Children-First Top-Down control strategy was used during the initial stages of solving a complex sub-problem, while a Depth-First Top-Down control strategy was used to solve simple sub-problems. Occasionally software engineers diverged from a Top-Down control strategy. There were eight types of divergences (such as working pursuing pre-requisites) that frequently occurred within the context of six different activities (such as making alternations to a solution).

The third study investigated the choice of local control strategies that supplement global control strategies. When software engineers were presented with a series of sub-problems to complete, they chose to complete the four sub-problems in the order that they were presented. The software engineers did not choose to pursue the series of sub-problems in any alternative order, such as the order of Easiest-First, or Analogies-First.

The fourth study investigated the environmental factors that influence the software engineers' choice of global control strategies. The study also investigated whether software engineers using the Breadth-First control strategy produced better results than software engineers using alternative control strategies. Three different text editors were produced to determine under what circumstance software engineers would use the Breadth-First control strategy. The editors also recorded the behaviour of the software engineers. The software engineers would only use the Breadth-First control strategy when the text editor forced them to follow the Breadth-First control strategy. However, the performance of the software engineers was better when they were forced to use the Breadth-First control strategy than when they used their preferred control strategies.
Acknowledgements

My appreciation for the numerous forms of assistance I received during the production of this thesis is owed to many. I would especially like to thank:

- Dr. Charles Crook (Director of Research),
- Dr. Tom Ormerod (External Supervisor, at Lancaster University),
- Dr. Andrew Shepherd (Internal Supervisor),
- Fellow students of Cognitive Ergonomics Research Group, and
- The volunteers who participated as subjects.

I would also like to thank the following people who provided me with research opportunities that supplied invaluable experience:

- Dr. David Gilmore (formerly of Nottingham University), and
- Prof. Anneliese von Mayrhofer (of Colorado State University).

Finally, thank you to friends and family for the support they gave without which this thesis would not have been possible.
Table of Contents

Abstract ............................................................................................................ i
Acknowledgements ........................................................................................... ii
Table of Contents .............................................................................................. iii
List of Figures .................................................................................................... xii
List of Tables ...................................................................................................... xiv

Chapter 1
Introduction ....................................................................................................... 1
  1.1. Aim ........................................................................................................... 3
  1.2. Overview ................................................................................................. 3
  1.3. The Software Industry ......................................................................... 4
  1.4. Problems With The Software Industry ................................................ 4
      1.4.1. Poor Productivity ....................................................................... 5
      1.4.2. Buggy Software ........................................................................ 6
      1.4.3. Solutions for the Software Industry .......................................... 7
          1.4.3.1. Programming Languages ............................................... 7
          1.4.3.2. Methodologies .............................................................. 7
          1.4.3.3. Computer Aided Software Engineering ....................... 8
          1.4.3.4. Summary ................................................................... 8
      1.4.4. An Alternative Approach to Solving the Problems of the Software Industry ..................................................... 8
  1.5. Designing Programs ............................................................................. 9
      1.5.1. Terms ......................................................................................... 9
          1.5.1.1. Hierarchical Relational Terms ....................................... 9
          1.5.1.2. Three Stages of Fulfilling A Goal .................................. 10
      1.5.2. Dimensions Involved In The Order of Goal Decomposition .......... 11
          1.5.1. Direction of Traversing ...................................................... 11
      1.5.2. Scheduling Groups of Sub-Goals (Global Control Strategies) ........ 12
          1.5.2.1. Breadth-First .............................................................. 12
          1.5.2.2. Depth-First .................................................................. 13
          1.5.2.3. Children-First .......................................................... 13
      1.5.3. Scheduling Individual Sub-Goals (Local Control Strategies) .......... 14
      1.5.4. Opportunism ............................................................................. 15
# Table of Contents

1.6. Rationale of Thesis .............................................................. 15

## Chapter 2

**Psychological Literature on Software Engineering** ...................... 17

2.1. Introduction ........................................................................... 19

2.2. Design .................................................................................. 19

2.2.1. Design as Problem Solving ............................................... 20

2.2.1.1. Types of Problems .................................................... 20

2.2.1.2. Information Processing .............................................. 20

2.2.2. Design Expertise ............................................................ 22

2.3. Domain Knowledge ............................................................... 24

2.3.1 Plans ................................................................................... 25

2.3.1.1. Rist's Model of Plans ................................................ 28

2.3.1.2. Procedural or Declarative Knowledge ......................... 29

2.3.1.3. Use of Plans outside of Pascal ..................................... 31

2.3.1.4. Summary of Plans ..................................................... 31

2.3.2. Analogies .......................................................................... 32

2.4. Heuristic Strategies ............................................................... 34

2.4.1. Formal Heuristic Strategies for Decomposing Problems ......... 34

2.4.2. Selection of Goal For Decomposition ................................. 35

2.5. Control Strategies ............................................................... 36

2.5.1. Global Control Strategies ............................................... 36

2.5.1.1. Factors That Affect Control Strategies ......................... 36

2.5.1.2. Structured Control Strategies for Designing Software ....... 39

2.5.1.3. Cue-Based Search ..................................................... 40

2.5.1.4. Opportunistic Behaviour ............................................ 41

2.5.2. Local Control Strategies .................................................. 45

2.5.3. Summary of Control Strategies ........................................ 46

2.6. Focus of Research ............................................................... 46

## Chapter 3

**Protocol Analysis** .................................................................... 47

3.1. Introduction ........................................................................... 49

3.1.1. Type of Action ............................................................... 49

3.1.2. Place of Action ............................................................... 50

3.1.3. Time of Action ............................................................... 51

3.1.4. Source of Data ............................................................... 51

3.2. Verbal Protocol Analysis ....................................................... 52
Table of Contents

3.2.1. Justification of Collecting Verbal Protocols ................................ 52
3.2.2. Collection of Verbal Data .................................................... 53
3.2.3. Treatment of Verbal Protocols ............................................ 53

3.3. Keystroke Protocol Analysis ................................................ 53
3.3.1. Collection of Keystrokes .................................................. 54
   3.3.1.1. Logger-2 .................................................................. 54
   3.3.1.2. TextRec .................................................................. 54
3.3.2. Data Collected .................................................................. 55
3.3.3. Coding of Keystroke Data ................................................ 55

3.4. Construction of Goal Hierarchy from Text of Software Solution .... 56
3.4.1. Ambiguity ........................................................................... 59

3.5. Combining Verbal and Keystroke Protocols ................................ 60
3.5.1. Combining Goal Hierarchies ............................................. 60
   3.5.1.1. Missing/Extra Goals ............................................... 60
   3.5.1.2. Different Arrangement of Goals .................................. 61
3.5.1.3. Altering the Description of the Goal’s Location ............... 62
3.5.2. Order of Goal Pursued ..................................................... 62

Chapter 4
The Global Control Strategies of Expert Prolog Programmers .......... 63

4.1. Introduction ........................................................................... 65
4.2. A Priori Advantages for Each Control Strategy ....................... 66
4.3. The Observational Study ...................................................... 68
   4.3.1. Rationale ..................................................................... 68
   4.3.2. Definition of Control Strategies ..................................... 69
4.4. Method ................................................................................ 70
   4.4.1. Participants .................................................................. 70
   4.4.2. Task and Materials ...................................................... 71
   4.4.3. Procedure .................................................................... 71
4.5. Results and Discussion ........................................................ 71
   4.5.1. Global Control Strategies ............................................. 71
   4.5.2. Structured-Divergent Activities ..................................... 74
      4.5.2.1. Transition to a node whose parent has not been coded, but which has been recognised .......... 75
      4.5.2.2. Transition that jumps back to a structured approach after a divergence ......................... 76
      4.5.2.3. Transition to amend an erroneous node ..................... 76
      4.5.2.4. Transition to capitalise on an analogy ...................... 76
## Chapter 5

### The Global Control Strategies of Expert C Programmers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. Introduction</td>
<td>85</td>
</tr>
<tr>
<td>5.1.1. Hypothesis</td>
<td>86</td>
</tr>
<tr>
<td>5.2. Method</td>
<td>87</td>
</tr>
<tr>
<td>5.2.1. Participants</td>
<td>87</td>
</tr>
<tr>
<td>5.2.2. Task and Materials</td>
<td>87</td>
</tr>
<tr>
<td>5.2.3. Procedure</td>
<td>87</td>
</tr>
<tr>
<td>5.3. Results and Discussion</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1. Global Control Strategies</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1.1. Comparison between the Global Control Strategies of C and Prolog Programmers</td>
<td>89</td>
</tr>
<tr>
<td>5.3.2. Divergence from the Global Control Strategies</td>
<td>90</td>
</tr>
<tr>
<td>5.3.2.1. Comparison of Opportunities between C and Prolog Programmers</td>
<td>90</td>
</tr>
<tr>
<td>5.4. When Do Opportunities Occur?</td>
<td>92</td>
</tr>
<tr>
<td>5.4.1. Prediction</td>
<td>92</td>
</tr>
<tr>
<td>5.4.2. Observation</td>
<td>93</td>
</tr>
<tr>
<td>5.4.3. A Secondary Analysis</td>
<td>94</td>
</tr>
<tr>
<td>5.4.3.1. Beginning to solve the problem</td>
<td>96</td>
</tr>
<tr>
<td>5.4.3.2. Beginning to solve a sub-problem</td>
<td>98</td>
</tr>
<tr>
<td>5.4.3.3. Changing the solution to a sub-problem</td>
<td>99</td>
</tr>
<tr>
<td>5.4.3.4. Analogy, reusing solution to a previous sub-problem</td>
<td>101</td>
</tr>
<tr>
<td>5.4.3.5. Debugging the solution to a sub-problem</td>
<td>102</td>
</tr>
</tbody>
</table>
# Table of Contents

5.4.3.6. No overall pattern to the Opportunistic Transitions .......................................................... 103

5.4.4. Isolated Opportunistic Transitions.......................................................................................... 104
  5.4.4.1. Debugging Transitions ....................................................................................................... 105
  5.4.4.2. Prerequisite Transitions .................................................................................................... 105
  5.4.4.3. Other Transitions ............................................................................................................. 106

5.4.5. Predicting the Next Opportunistic Transition ........................................................................ 106
  5.4.5.1. The Debugging-Debugging Pairing ...................................................................................... 107
  5.4.5.2. The Prerequisite-Prerequisite Pairing ................................................................................. 107
  5.4.5.3. The Prerequisite-Jump Back Pairing .................................................................................. 107

5.5. Discussion ................................................................................................................................. 107
  5.5.1. Factors Effecting the Frequency of Opportunistic Transitions ................................................... 108
  5.5.2. The Nature of the Problem Task .......................................................................................... 109
  5.5.3. Advantages of the Breadth-First Control Strategy ................................................................... 110

Chapter 6

Local Control Strategies .................................................................................................................. 111

6.1. Introduction ............................................................................................................................... 113
  6.1.1. Local Control Strategy is Needed to Supplement Global Control Strategies .......................... 113
  6.1.2. Opportunism and Local Control Strategies .......................................................................... 113
  6.1.3. Possible Local Control Strategies ......................................................................................... 114
  6.1.4. Hypothesis ............................................................................................................................ 115

6.2. Method ....................................................................................................................................... 116
  6.2.1. Design ................................................................................................................................... 116
  6.2.2. Participants ............................................................................................................................ 116
  6.2.3. Materials ............................................................................................................................... 116
  6.2.4. Procedure .............................................................................................................................. 117

6.3. Results ........................................................................................................................................ 118
  6.3.1. Measurement of Local Control Strategies ............................................................................. 118
  6.3.2. Verbal Reports ....................................................................................................................... 120
  6.3.3. Measurement of Difficulty ..................................................................................................... 121

6.4. Discussion .................................................................................................................................. 122
  6.4.1. Is Local Control Strategy Determined by Dependence or Text Order? ................................. 123
  6.4.2. Local Control Strategy for Richer Problems .......................................................................... 123
  6.4.3. Is a Global Control Strategy Opportunistic? ....................................................................... 124
## Table of Contents

### Chapter 7

**Using the Breadth-First Global Control Strategy**

- **7.1. Introduction** .......................................................... 128
  - **7.1.1. Parsing Required for Each Global Control Strategy** .................. 128
    - **7.1.1.1. Examination of Parsing Required for Each Global Control Strategy** ................................................................. 129
  - **7.1.2. Exacerbating the Parsing Problem** ....................................... 132
  - **7.1.3. An Additional Problem with the Breadth-First Global Control Strategy** .......................................................... 133
  - **7.1.4. The Aims of the Study** .................................................. 134
  - **7.1.5. Hypothesis** ....................................................................... 134

- **7.2. Method** ............................................................................ 134
  - **7.2.1. Participants** ............................................................. 134
  - **7.2.2. Task** ..................................................................... 135
  - **7.2.3. Material** ................................................................ 135
  - **7.2.4. Procedure** ............................................................... 137

- **7.3. Results and Discussion** ......................................................... 138
  - **7.3.1. Conforming to the Breadth-First Global Control Strategy** .......................................................... 138
  - **7.3.2. Comparing the Performance of Programmers using the Breadth-First Global Control Strategy against those who did not** .................................................................. 140
    - **7.3.2.1. Completion of Solution** ........................................ 140
    - **7.3.2.2. Errors in Solution** ................................................. 141
    - **7.3.2.3. Movement between Procedures** .................................. 144
    - **7.3.2.4. Rate of Construction of Sub-Problems** .................... 145
  - **7.3.3. Summary** ....................................................................... 146

### Chapter 8

**General Discussion** ...................................................................... 147

- **8.1. Aim of Research** .......................................................... 149
- **8.2. Review of Research** .......................................................... 149
  - **8.2.1. Review of Methodology** .................................................. 149
    - **8.2.1.1. Concurrent Verbal Reports** ...................................... 150
    - **8.2.1.2. Text Editors** .......................................................... 150
  - **8.2.2. Review of Behaviour of Prolog Programmers** ..................... 151
    - **8.2.2.1. Global Control Strategies** ....................................... 151
    - **8.2.2.2. Transitions Not Conforming to a Global Control Strategy** .......................................................... 152
8.2.3. Review of Behaviour of C Programmers........................................ 152
  8.2.3.1. Global Control Strategies.................................................. 152
  8.2.3.2. Frequency of Opportunistic Transitions............................... 153
  8.2.3.3. Occurrence of Opportunistic Transitions............................... 153
8.2.4. Review of Local Control Strategies........................................... 154
8.2.5. Review of Encouraging Breadth-First Global Control Strategy.................. 155

8.3. Comparison of Results With The Literature...................................... 156
  8.3.1. Comparison with Opportunistic Studies..................................... 157
  8.3.2. Comparison with Structured Studies........................................ 157
  8.3.3. Comparison with Rist's Cue-Based Search Theory............................ 158
  8.3.4. An Extension to the Parsing-Gnisrap Model................................ 159

8.4. Implications of Results For Industry............................................. 159
  8.4.1. Training of Programmers..................................................... 159
  8.4.2. Design of Software Tools.................................................... 160

8.5. Issues for Further Investigations................................................. 161
  8.5.1. Factors Influencing Choice of Control Strategy............................ 161
    8.5.1.1. Object-Oriented Languages............................................. 161
    8.5.1.2. Cognitive Resources.................................................... 162
    8.5.1.3. Expertise of Programmers............................................. 162
  8.5.2. Most Effective Control Strategy............................................ 162
  8.5.3. Improving Construction of Software........................................ 163

8.6. Critique of Research Methodology............................................... 163
  8.6.1. Weaknesses of Research Methodology....................................... 163
  8.6.2. Implications of Weaknesses for Results.................................... 164
    8.6.2.1. Only one task was used in each studies............................... 164
    8.6.2.2. The problems used were not large scale.............................. 164
    8.6.2.3. Few participants....................................................... 165
    8.6.2.4. Motivation............................................................... 165
    8.6.2.5. Presentation and Execution Order Being Identical.................... 165
    8.6.2.6. Similarity of Ease...................................................... 166
    8.6.2.8. Viscosity of Environment.............................................. 166
  8.6.3. How to Improve Research Methodology....................................... 167
    8.6.3.1. How to Improve Generic Weaknesses................................... 167
    8.6.3.2. How to Improve Weaknesses of the Local Control Strategies Study...... 167
    8.6.3.3. How to Improve Weaknesses of the Breadth-First Study................ 167
# Table of Contents

8.7. Conclusion .......................................................................................... 168

**Appendix A**
Decomposition of a Hypothetical Goal Hierarchy (shown in Figure 1.4) Using Different Global Control Strategies .................................................. 171

**Appendix B**
Tables for Protocol Analysis ................................................................. 174

  B.1. Generalised Actions ................................................................. 174

  B.2. Structure of TextRec Log Files .................................................. 175

    B.2.1. Parameters Associated with Each Type of Interaction ............. 176

**Appendix C**
Materials Used in Global Control Strategies of Prolog Programmers Study ................................................................. 177

  C.1. Standard Introduction .............................................................. 177

  C.2. Specification of Practice Task .................................................... 178

  C.3. Specification of the Signals Problem ............................................ 179

**Appendix D**
Activity Transition Depths for Ss A1, A2 and A3 .................................. 181

**Appendix E**
Materials Used in Global Control Strategies of C Programmers Study ................................................................. 183

  E.1. Standard Introduction .............................................................. 183

  E.2. Specification of Practice Task .................................................... 184

  E.3. Specification of Main Task ........................................................ 185

**Appendix F**
Occurrence of Opportunism ................................................................. 187

**Appendix G**
Materials Used in Local Control Strategies Study ................................... 189

  G.1. Introduction .............................................................................. 189

  G.2. Program To Be Completed ........................................................ 190

  G.3. Questionnaire .......................................................................... 191

**Appendix H**
Materials Used in Encouraging Use of Breadth-First ................................ 192

  H.1. Standard Introduction .............................................................. 192
Table of Contents

H.2. The Editor (given to participants using the neutral editor) .................. 193
H.3. The Editor (given to participants using the supportive editor) .......... 194
H.4. The Editor (given to participants using the enforcing editor) .......... 195
H.5. The C language ............................................................................. 196
H.6. Specification for Main Task .................................................... 197
H.7. Questionnaire ............................................................................. 198

Appendix I

Example of Analysis of a Prolog Programmer (A4) ....................... 199
I.1. Final Verbal & Keystroke Protocol of A4 .................................... 199
I.2. Prolog Solution of A4 .......................................................... 231
I.3. Goal Hierarchy of A4 ........................................................... 235
I.4. Order In Which A4 Constructed His Goal Hierarchy ............... 242

Appendix J

Individual Results and Statistical Tests........................................... 248
J.1. Comparison of Order of Sub-Problems ..................................... 248
J.2. Complexity of Solution for Sub-Problems ................................... 250
J.3. Time Taken to Solve Sub-Problems ........................................... 251
J.4. Percentage of Transitions Devoted to Each ................................ 252
J.5. Time Taken to Complete Solution By Order of Being Solved .... 253
J.6. Mean Time Spent Delibrating at Start of a Procedure .......... 254
J.7. Mean Time Spent between each Keystroke .............................. 255
J.8. Number of Errors in a Solution................................................. 256
J.9. Amount of a Solution Completed .............................................. 257
J.10. Movements of each Participant between Procedures ............. 258

References ................................................................................. 259
List of Figures

Figure 1.1 Time Taken to Complete a Project According to its Size ..................... 5
Figure 1.2 Pictorial Representation of Hierarchical Relationships ......................... 10
Figure 1.3 Direction of Traversing ........................................................................ 11
Figure 1.4 A Hypothetical Design Goal Hierarchy .............................................. 12
Figure 1.5 Goal Recognition Order With A Breath-First Strategy ......................... 12
Figure 1.6 Goal Recognition Order With A Depth-First Strategy ........................... 13
Figure 1.7 Goal Recognition Order With A Children-First Strategy ....................... 13
Figure 2.1 Neisser's Model of Cognition .......................................................... 21
Figure 2.2 Three Ways in which a Cake can be Divided into Four Equal Portions ....... 23
Figure 2.3 A Node with Linear, Control Flow, and Data Flow Arcs ....................... 29
Figure 2.4 Stages of Reusing Solution .................................................................. 33
Figure 2.5 Order of Depth-First Construction (left), Breadth-First (right) ............. 39
Figure 3.1 Example of Enumerated Nodes ............................................................ 50
Figure 3.2 Example of Hierarchical Goal Structure ............................................ 59
Figure 3.3 Verbal Protocol Goal Hierarchy ......................................................... 60
Figure 3.4 Keystroke Protocol Goal Hierarchy .................................................... 61
Figure 3.5 Combined Protocol Goal Hierarchy ..................................................... 61
Figure 4.1 Observed Depth at which A4 Worked across Transitions ..................... 73
List of Figures

Figure 4.2
Hypothetical Decomposition using a Breadth-First Control Strategy .......... 73

Figure 4.3
Hypothetical Decomposition using a Children-First Control Strategy .......... 74

Figure 5.1
Possibility for Opportunism at Stages of Design of Solution ..................... 92

Figure 5.2
Opportunism at Stages of B1’s Design of Solution .................................. 93

Figure 5.3
Opportunism at Stages of B3’s Design of Solution .................................. 94

Figure 5.4
Clusters of Opportunism ................................................................. 95

Figure 5.5
The Initial Design Activity of Designer B5 .......................................... 98

Figure 5.6
Designer B1 Amending his Solution to the Design Problem ........................ 100

Figure 8.1
A Tree and a Net ........................................................................ 161

Figure 8.2
A Net and its Spanning Tree ............................................................ 162

Figure D.1
A1’s Depth of Activity Transitions .................................................. 181

Figure D.2
A2’s Depth of Activity Transitions .................................................. 181

Figure D.3
A3’s Depth of Activity Transitions .................................................. 182

Figure F.1
Opportunism at Stages of B2’s Design of Solution ................................. 187

Figure F.2
Opportunism at Stages of B4’s Design of Solution ................................. 187

Figure F.3
Opportunism at Stages of B5’s Design of Solution ................................. 188
List of Tables

Table 4.1
Functional Definitions of Activity Transition Conforming to a Structured Approach ........................................................................................................................................... 69

Table 4.2
Conservative Measure of Moves Conforming to Structured Control Strategies .............................................................................................................................................................. 72

Table 4.3
Combined Conservative and Liberal Measure of Moves Conforming to Structured Control Strategies .............................................................................................................................................................. 72

Table 4.4
Frequency and Absolutes for each Type of Structure-Divergent Transitions ........................................................................................................................................................................................................ 75

Table 5.1
Comparison of Prolog and C .............................................................................................................................................................................................................................................. 86

Table 5.2
Percentage of Moves of C Programmers Conforming to Structured Control Strategies ........................................................................................................................................................................................................ 88

Table 5.3
Percentage of Transitions Conforming to Children-First and Depth-First Control Strategies ........................................................................................................................................................................................................ 89

Table 5.4
Occurrence of Various Opportunistic Transitions .................................................................................................................................................................................................................................. 90

Table 5.5
Opportunities of C and Prolog Programmers ...................................................................................................................................................................................................................................... 91

Table 5.6
Opportunistic Activities .................................................................................................................................................................................................................................................. 96

Table 5.7
Occurrence of Opportunistic Transitions in Clusters and in Isolation ......................................................................................................................................................................................................... 103

Table 5.8
Number of Opportunism in Clusters and in Isolation .................................................................................................................................................................................................................................. 104

Table 5.9
Occurrence of types of Opportunism following previous Opportunism ......................................................................................................................................................................................................................... 106

Table 6.1
Presentation and Difficulty of Sub-Problems .............................................................................................................................................................................................................................................. 118

Table 6.2
Prediction of Proposed Local Control Strategies .............................................................................................................................................................................................................................................. 119
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Self Reported Ease Compared Against Quantified Ease</td>
<td>121</td>
</tr>
<tr>
<td>6.4</td>
<td>Time Taken to Solve First-Fourth Sub-Tasks</td>
<td>122</td>
</tr>
<tr>
<td>7.1</td>
<td>Movement between Procedures</td>
<td>139</td>
</tr>
<tr>
<td>7.2</td>
<td>Completion of Task</td>
<td>141</td>
</tr>
<tr>
<td>7.3</td>
<td>Occurrence of Errors for each Control Strategy</td>
<td>142</td>
</tr>
<tr>
<td>7.4</td>
<td>Time Spent Deliberating</td>
<td>144</td>
</tr>
<tr>
<td>7.5</td>
<td>Time between Keystrokes</td>
<td>145</td>
</tr>
<tr>
<td>A.1</td>
<td>Breadth-First Global Control Strategy Decomposition</td>
<td>171</td>
</tr>
<tr>
<td>A.2</td>
<td>Depth-First Global Control Strategy Decomposition</td>
<td>172</td>
</tr>
<tr>
<td>A.3</td>
<td>Children-First Global Control Strategy Decomposition</td>
<td>173</td>
</tr>
<tr>
<td>B.1</td>
<td>Protocol Analysis' Codes</td>
<td>174</td>
</tr>
<tr>
<td>B.2</td>
<td>List of Interactions Recorded By TextRec</td>
<td>176</td>
</tr>
<tr>
<td>I.1</td>
<td>Combined Verbal &amp; Keystroke Protocol of A4</td>
<td>199</td>
</tr>
<tr>
<td>I.2</td>
<td>Comparing Transition against Rules of Structured Control Strategies</td>
<td>242</td>
</tr>
<tr>
<td>J.1</td>
<td>Key to Contents of the Cell's in Table J.2</td>
<td>248</td>
</tr>
<tr>
<td>J.2</td>
<td>Testing Correlation Between Expected and Observed Order of Sub-Problems using Kendall’s τ</td>
<td>249</td>
</tr>
<tr>
<td>J.3</td>
<td>Testing tread of Complexity increase with Difficulty of Sub-Problem using Page’s L Tread Test for Related Samples</td>
<td>250</td>
</tr>
<tr>
<td>J.4</td>
<td>Testing tread of Time increase with Difficulty of Sub-Problem using Page’s L Tread Test for Related Samples</td>
<td>251</td>
</tr>
</tbody>
</table>
List of Tables

Table J.5
Testing tread of Transition increase with Difficulty of Sub-Problem using Page's L Tread Test for Related Samples .................................................. 252

Table J.6
Testing tread of Transition increase with Difficulty of Sub-Problem using Page's L Tread Test for Related Samples .................................................. 253

Table J.7
Independent T Test - difference between 'Mean Deliberation at Start of a Procedure' for Strategies Used ................................................................. 254

Table J.8
Independent T Test - difference between 'Mean Time between Keystrokes' for Strategies Used ................................................................. 255

Table J.9
Independent T Test - difference between 'Number of Errors in Solution' for Strategies Used ................................................................. 256

Table J.10
Independent T Test - difference between 'Amount of Solution Completed' for Strategies Used ................................................................. 257

Table J.11
Movements of each Participant between Procedures Movements of each Participant between Procedures .................................................. 258
Chapter 1

Introduction
Contents of Chapter

This chapter describes the aim of the thesis, and an overview of how this aim is to be achieved. The chapter describes the problems facing the software industry, namely poor productivity of software engineers and poor quality of software, and how these problems are likely to increase as the capacity of computers continue to increase. The chapter argues that these problems can successfully be elevated by empirically investigating the cognitive processes of software engineers and how to support their cognitive processes. The chapter provides a description of software, and the order in which software is constructed. The terms introduced will facilitate the interpretation of behaviour of software engineers who have been observed in the experiments described in this thesis.
1. Introduction

1.1. Aim

The aim of the research is to investigate (1) the order in which software engineers construct software solutions; (2) the reasons why software engineers select a particular strategy for determining order in which solutions are designed (such strategies are called control strategies); and (3) the effectiveness of the various control strategies. The purpose of the aims is to provide a basis to help determine which control strategies provide the best software engineering performance, and how it is possible to encourage the use of the best control strategies. The paradigm of the investigation is cognitive psychology. This discipline was selected as it is recognised that the software engineer is a key factor that determines the usefulness of a control strategy. Cognitive psychology can be used to explain why a particular control strategy is difficult to use, and how to reduce its cognitive effort. The reduction of cognitive effort can result in a reduction of induced error and a reduction of time taken to complete a task. Cognitive psychology can be applied to software engineering tasks to improve the productivity of developing software and to improve the quality of emerging software.

1.2. Overview

The thesis is composed of three parts (1) Background Information; (2) Empirical Studies; and (3) Conclusion.

The first part, Background Information, contains three chapters. Chapter 1, the introduction, provides a description of the software industry, the problems facing the software industry, and an explanation of software structure. Chapter 2, the literature review, outlines previous psychological examination of the mental models and cognitive processes involved in constructing a software solution. Chapter 3, the protocol analysis, describes the method of data analysis used in the empirical studies to aid the interpretation of the data collected.

The second part, Empirical Studies, describes four studies examining the order in which software solutions are constructed. Each study is described in a separate chapter. Chapter 4 examines the global control strategies of Prolog programmers, and incidents where programmers do not conform to a global control strategy. Chapter 5 details a replication of the previous study except that the participants were C programmers. In addition, Chapter 5 examines where and why divergence from a global control strategy occurs. Chapter 6 describes a study into the choice of local control strategies. Chapter 7 examines how to encourage software engineers to use the Breadth-First control strategy and whether performance improves when software engineers use the Breadth-First control strategy.
The final part, Conclusion, is discussed in Chapter 8. Chapter 8 summarises and critiques the previous findings described in Chapters 4 to 7, and discusses the implication of the findings for the software industry and for the literature on psychology of programming. Finally, Chapter 8 outlines questions relating to the topic covered by this thesis but are beyond the bounds of the thesis, and possible studies for providing answers to those questions.

1.3. The Software Industry

The software business is based in a comparatively new engineering discipline. The first electronic computers were built in the 1940's. During the last fifty years the hardware industry and the software industry have expanded at an astonishing rate. For example, Moore's Law observes that the MIPS\textsuperscript{1}/cost and memory/cost ratios double approximately every 18 months (Moore, 1965). The memory capacity of computers increased as memory became cheaper. In 1980, the ZX-80 was released with 1 kilobyte of memory. In 1997, a typical personal computer would rarely have less than 8 megabytes of memory, and more often have 16 or more megabytes of memory. Within 17 years, the capacity of a personal computer's memory has increased by a factor of between 8192 and 16384, without an increase of costs.

An effect of the increased availability of memory is for software to expand to capitalise on the extra storage space (DeMarco & Miller, 1996). With more memory, software has become more complex. This has made it possible for an increasing range of activities to be supported by computers. For example in the past a personal computer would be able to support a simple text editor, with very few functions. As the size of memory increased, the size and functionality of software have also increased. So that text editors evolved into word processors, then they became WYSIYWYG\textsuperscript{2} word processors, and then they evolved into DTP\textsuperscript{3} software.

1.4. Problems With The Software Industry

Unfortunately as software comes ever larger, software problems also become larger. Some problems become disproportionately larger (Petroski, 1994, p. 29). There are essentially two problems confronting the software industry, these are:

- Poor Productivity, and
- Buggy Software.

---

\textsuperscript{1} MIPS – Million Instructions Per Second (speed of processing commands).
\textsuperscript{2} WYSIYWYG – What You See Is What You Get (i.e. the monitor’s output matches the printer’s output).
\textsuperscript{3} DTP – Desk-Top Publishing.
1.4.1. Poor Productivity

Brooks (1975) wrote several essays on the experience of software engineers working on large projects. A common experience is that as the size of a project increase, the amount of man-months involved in the project increases geometrically.

![Figure 1.1: Time Taken to Complete a Project According to its Size](image)

This observation simply means that as software becomes larger, the productivity of each engineer decreases. A tempting solution is to increase the number of engineers involved in a project. Such a solution is fraught with difficulties as productivity of each worker is also dependent on the number of workers involved. Increasing the number of workers can decrease the productivity of each worker. This is a result of the increased amount of communications that is required.

Since man-months are geometrically proportional to the size of a project, and there is a diseconomy of scales relating the number of engineers that can be employed on a project, this entails that the time needed to complete a project is geometrically proportional to the size of a project, indeed this has been observed and reported by Gibbs (1994).

Not only do clients have to wait longer for the completion of large software, they also have to pay for the explosion of man-months. The costs are exacerbated by the inability of project managers to estimate the time needed to complete a project.

"The average software development project overshoots its schedule by half; larger projects generally do worse" – (Gibbs, 1994).

The rates of productivity, and the costs of production are only part of the problem. Poor quality of products is also a problem. For every six new large-scale projects that

---

4 Man-months is the sum of time each engineer spends on a project.
become operational, two others are cancelled. Also three-quarters of all large projects produce software that does not function as intended (Gibbs, 1994).

The causes of poor productivity are many and varied. Causes include increase in communications between engineers, and accordingly increases in misunderstanding between engineers. Clients also have a poor understanding of their requirements, some of which may be in conflict with each other. A major cause is the increased number of errors that occur in a program, and the difficulty of finding some of the errors.

1.4.2. Buggy Software

Errors in software are another set of problems that encounters the software industry. The occurrence of errors increases as the size of the program increases. Also discovering errors become more difficult as the size of the program increases.

The discovery of an error within a program before it is released, extends the man-months necessary to complete the project, and as a result increases the cost of the project both for the client and for the manufacturer. Errors in released software can have dramatic effects for the users.

As software is being used to support more complex tasks, errors can increasingly be bizarre, costly and even deadly. An example of a bizarre bug is a F16 simulation. Whenever the plane crosses the equator, it turns upside (SIGSOFT, 1980). The 1996 explosion of the European space rocket ARIANE 5 was the result of a software error (Lions, 1996). The cost of the explosion includes the cargo worth £330 million and the launcher worth £113 million (Highfield, 1996). The cargo was not insured (Flower, 1996). The most worrying errors occur in software that are meant to reduce risk to human life. Monday, 26 October 1992, the London Ambulance Service introduced a new software based ambulance dispatch system. Unfortunately the software was not adequate to the task, and an estimated 20 people died as a result of ambulances arriving too late (MacKinnon & Goodwin, 1992).

As well as the potential for errors to become more damaging, errors are becoming more difficult to detect. Complex software typically has to cope with a potentially wide range of situations. In some cases there are an infinite set of inputs a system can receive. This makes it impractical and sometimes impossible to test the full range of situations in which a system has to work. Consequentially, software engineers have to select a sub-set of possible input data that is representative of all possible situations. At the moment there is no hard mathematical model to guide software engineers in their choice of test data. This inevitably leads some situations not being represented by the test data, such as with the plane in the F16 simulation being flown across the equator.
1.4.3. Solutions for the Software Industry

Computer scientists and software engineers have made attempts to improve the productivity and the quality of the design and implementation of software. The main approach has been to make the process of software engineering more rigorous. This encompasses many activities. The success of this approach is difficult to evaluate, since evidence of gains is primarily anecdotal, and rarely collected scientifically.

The suggestions for improvements include rules-of-thumb (McConnell, 1993, is a compilation of such rules); using particular programming languages; using particular methodologies; and using CASE\textsuperscript{5} tools.

1.4.3.1. Programming Languages

A large collection of computer programming languages have been produced. The designers of each language claim that using their particular language is more beneficial than using any other language when solving particular problems. The basis of some claims is that the language complements a class of methodologies; for example, C++ supports object-oriented methodologies.

Other languages make claims that their level of description will improve productivity. For example, the Prolog level of description is supposed to be similar to that of problem specifications, so that programming becomes almost a simple automatic process of converting each sentence in the specification into a relational statement (Malpas, 1987). The users of Prolog should not need to spend a large proportion of their time coding. Furthermore the occurrence of implementation errors should be negligible.

1.4.3.2. Methodologies

At the moment there are five classes of methodologies: process, data, object, semantic, or none (Conger, 1994). For each class there are many methodologies that have been designed (e.g. SADT\textsuperscript{6}, JAD\textsuperscript{7}, ERD\textsuperscript{8}, etc.), and some methodologies cover several classes, such as SSADM\textsuperscript{9} (Ashworth & Goodland, 1990). The various methodologies essentially prescribe the procedures, techniques and processes by which a software engineering goal can be achieved. Each class of methodology identifies an aspect by which to view the software problem. For example, process methodologies emphasise the flow of data.

---

\textsuperscript{5} CASE - Computer Aided Software Engineering.
\textsuperscript{6} SADT - Structured Analysis Design Technique.
\textsuperscript{7} JAD - Joint Application Design.
\textsuperscript{8} ERD - Entity Relationship Diagram.
\textsuperscript{9} SSADM - Structured Systems Analysis and Design Method.
1.4.3.3. **Computer Aided Software Engineering**

Software engineering methodologies often involve simple tasks, as well as some more complex tasks. In order to allow engineers to focus on the complex tasks, the simple tasks (e.g. drawing neat diagrams) can be automated and given to CASE applications to complete. The ultimate CASE tool would automate the whole of software engineering (McClure, 1989). Currently, the spending on CASE tools has been conservatively estimated to be growing at 19% annually (Banker & Kauffman, 1990). CASE tools also maintain the integrity of the solutions. McClure (1989) produced the following list of potential benefits to be gained from using CASE tools.

- Makes structured techniques practical.
- Enforces software/information engineering.
- Improves software quality through automated checking.
- Makes prototyping practical.
- Simplifies program maintenance.
- Speeds up development process.
- Frees developer to focus on creative part of software development.
- Encourages evolutionary and incremental development.
- Enables reuse of software components.

However if a CASE tool is poorly designed, the effort of using that tool may exceed the effort required to perform manually the tasks that were given over to the tool.

1.4.3.4. **Summary**

In 1991, the Capability Maturity Model (CMM) was produced by the Software Engineering Institute, who are funded by the US military. CMM measures, on a scale from one to five, how methodically a software team develops software. In 1994, 75% of teams were evaluated at level one, i.e. chaotic (Gibbs, 1994). Less than 1% of teams reached level five. Loral who are responsible for the on-board software of NASA’s space shuttle are a level one company. Loral produces far fewer errors than most software teams (McConnell, 1993). This suggests that employing methodologies improves the quality of software. However, it is not clear which methodologies are superior for given situations.

1.4.4. **An Alternative Approach to Solving the Problems of the Software Industry**

Computer scientists and software engineers’ proposals for improving productivity and quality have been guided either by practical experience or by mathematical theories. Software teams that employ the proposals tend to achieve better results than those that do not. However, the up-take of the various strategies is slow, and it is not clear which particular strategy is superior for a given problem.
An important factor that governs the usefulness of a proposal is whether it is congruent with cognitive processes of the programmer. By examining the cognitive processes of software engineers it is possible to evaluate the tasks which they find easy and which they find difficult. Cognitive psychology can also model the processes involved in completing a task. By identifying the difficult tasks, and how these tasks are achieved, it is possible to propose means for simplifying the difficult tasks, by reducing the cognitive effort required to complete the tasks. Simplifying difficult tasks has the effect of reducing mistakes, and reducing the time taken to complete a task. Thus cognitive psychology can provide an empirical method for evaluating software engineering strategies. The results of empirical evaluation can be applied to improve the productivity of software development process and the improve the quality of software.

The aim of this thesis is to report studies examining and evaluating particular software engineering strategies for designing and implementing software solutions.

1.5. Designing Programs

A computer program typically contains a main goal that is defined in terms of simpler goals. These goals are further simplified, the process continues until a goal is reached which can be described by a single line of code.

The act of programming involves constructing a hierarchy of goals (Wirth, 1971). The building of a program can be seen as an analogy to traversing a tree, where every node in the tree must be accessed and considered.

1.5.1. Terms

In order to discuss the process of designing programs it is first necessary to introduce a number of terms that describe the relationship between two nodes within a hierarchy; and terms that describe the process of designing a solution of a goal presented by a node.

1.5.1.1. Hierarchical Relational Terms

Since the structure of a program is hierarchical, and families are hierarchical, it is helpful to apply terms of family relationships to describe the relationships between program segments. The terms of male relationships have been applied where no neutral term exists.

- **offspring** - offspring $X$ of a node $Y$, is a node where it is a subdivision of $Y$. The amalgamation of any subset of offspring cannot form an offspring.
- **parent** - a node $X$ is parent of node $Y$, when $Y$ is an offspring of $X$.
- **sibling** - two different nodes are siblings when they share the same parent.
- **ancestor** - $X$ is an ancestor of $Y$, when either $X$ is a parent of $Y$, or $X$ is a parent of an ancestor of $Y$.
- **nephew** - $X$ is a nephew of $Y$, when $X$ is a offspring of $Y$'s sibling.
• uncle - X is a uncle of Y, when Y is a nephew of X.
• cousin - X is a cousin of Y, when the parent of X is a sibling of Y’s parent.
• nth cousin - X is a nth cousin of Y, when the parent of X is a (n - 1)th cousin of Y’s parent. A first cousin is the same as a cousin.
• peer - X is a peer of Y, when X is a nth cousin of Y or a sibling of Y, i.e. both X and Y are off the same generation.

![](image)

Figure 1.2: Pictorial Representation of Hierarchical Relationships

1.5.1.2. Three Stages of Fulfilling A Goal

In any strategy of designing a program each (sub-)goal of the completed program must pass through three stages. The three stages are Recognise, Describe and Complete. The meaning of each term is given below.

• Recognise a goal. A goal is recognised when it is realised it must be completed in order for the program problem to be solved (i.e. the goal has been recognised as a necessary part of the overall program).

• Describe a goal. A goal is said to have been described once all of it immediate sub-goals have been recognised.

• Complete a goal. A completed goal has had all of its descendants recognised. It follows that a goal is completed when all its immediate sub-goals have been recognised and completed. (A completed goal is one that has fully designed, and does not necessarily entail as having been implemented.)

Simple goals which cannot be further simplified into sub-goals are considered to have been described and completed as soon as they have been recognised.
1.5.2. Dimensions Involved In The Order of Goal Decomposition

An overview of the way in which the program tree can be traversed is given below. There are three main dimensions to goal directed traversal strategies. In addition there is the possibility of Opportunistic behaviour, which is data driven rather than being driven by the structure of the current state of the goal hierarchy.

- **Direction of traversing.** There are two dimensions in which to traverse a tree. The first is up and down (levels of abstraction) and the second is left and right (start to end).
- **Scheduling groups of sub-goals** is the strategy used to transverse the tree in a general direction. This dimension describes how the sub-goals of a goal are scheduled as a group. In this thesis strategies for scheduling groups of sub-goals are called **global control strategies**.
- **Scheduling individual sub-goals** is the order in which a problem’s sub-goals belonging to the same group are scheduled. In this thesis, strategies for scheduling individual sub-goals are called **local control strategies**.
- **Opportunism.** The previous three dimensions view the construction of a tree as being directed by goals that have been scheduled. However, there are situation in which the next goal to be described is guided by data that had recently been discovered. Opportunism is data driven behaviour.

![Figure 1.3: Direction of Traversing](image)

1.5.1. Direction of Traversing

There are two dimensions in which a tree can be traversed. The first dimension is between the abstract and the concrete. Programmers can either work Top-Down (abstract to concrete) or Bottom-Up (concrete to abstract). The second dimension is between the start and the end of a the problem. Programmers working forwards begin at the start of the problem and finish at the end, while programmers working backwards begin at the end of the problem and finish at the start.
1.5.2. Scheduling Groups of Sub-Goals (Global Control Strategies)

There are three strategies for scheduling groups of sub-goals, these are:

- Depth-First,
- Breadth-First,
- Children-First.

The first two are commonly recognised in the literature on prescriptive design. The third, Children-First, is introduced in this thesis. Each of the three approaches does not make specific predictions about the order in which the sub-goals in a group are described, that is a separate question addressed in section 1.5.3.

The application of a global control strategy leads to the creation of a stack consisting of goals that have been recognised and need to be described. Goals are removed from the goal stack once they have been described. The goal stacks created using the three control strategies to decompose the hierarchy shown in Figure 1.4 are given in Appendix A.

Below is a description of the three global control strategies. In each section is an example of the order in which the strategy would decompose the hypothetical goal hierarchy shown in Figure 1.4.

1.5.2.1. Breadth-First

A Breadth-First strategy (illustrated in Figure 1.5) specifies that all the goals of a particular level in the hierarchy must be described before the designer describes any of the goals of a lower level.

---

Figure 1.4: A Hypothetical Design Goal Hierarchy

Figure 1.5: Goal Recognition Order With A Breadth-First Strategy
1.5.2.2. Depth-First

A Depth-First strategy (illustrated in Figure 1.6) reflects the pursuit to completion of a goal as soon as that goal has been recognised (i.e. the designer endeavours to develop that goal’s hierarchy until simple sub-goals are reached before exploring that goal’s siblings).

![Figure 1.6: Goal Recognition Order With A Depth-First Strategy](image)

Depth-First and Breadth-First strategies are recognised in the computer science literature (e.g. Wirth, 1971) as being Top-Down structured approaches, as the programmer recognises, describes and completes each node in a design goal hierarchy in a systematic order determined by the chosen control strategy and the structure of the hierarchy.

1.5.3.3. Children-First

Whilst Breadth-First and Depth-First strategies are recognised as the main structured control strategies, there is another possible method which I have termed a Children-First approach (illustrated in Figure 1.7).

![Figure 1.7: Goal Recognition Order With A Children-First Strategy](image)

Superficially, Children-First appears to be a mixture of the Depth-First and the Breadth-First approaches. The Children-First strategy is applied recursively, as follows. First the goal is recognised, then the programmer proceeds to describe the goal by recognising all of its immediate sub-goals. Then, one of the sub-goals is selected and the above steps are applied until that sub-goal has been completed, at which point, the programmer selects the next sub-goal to complete, and so on. This continues until the programmer has completed all the sub-goals, and thus has completed the goal itself.
1.5.3. Scheduling Individual Sub-Goals (Local Control Strategies)

In the previous section the scheduling of groups of sub-goals was discussed. What follows is an examination how the sub-goals within these groups are individually scheduled. When the programmer selects a sub-goal group he must choose the order in which to describe the sub-goals. Typically this ordering is carried out implicitly; the programmer chooses the first sub-goal to describe, and when this has been described the programmer chooses the next sub-goal to describe. There is a cycle of choosing a sub-goal and describing it as opposed to choosing the order in which to pursue all sub-goals then describing the sub-goals in that order.

There have been many suggestions for the criteria by which the next sub-goal to be described is chosen (Visser, 1990; Ball & Ormerod, 1995), but there have been few empirical studies on the strategies involved in the scheduling of goals. Listed below is a description of a few of the suggestions.

- **First-Mentioned.** The control strategy of first-mentioned is often used in conjunction with the Breadth-First control strategy. The first sub-goal mentioned in the group is the first one to be described. The second sub-goal mentioned is then the next to be described. This continues until all the sub-goals have been described.

- **Easiest-First.** A programmer who uses this control strategy selects the sub-goal that he considers to be the easiest to describe. Part of the reason behind this strategy is that describing one sub-goal will make the remaining sub-goals easier to describe. So the perceived net result of Easiest-First is that it is the easiest method for describing a group of sub-goals.

- **Most-Difficult-First.** This local control strategy enables a programmer to explore the feasibility of a solution to the software design problem. If programmers are not able to complete the most-difficult sub-problem, then they are not able to complete the overall problem. By first checking the feasibility of the most-difficult sub-goal, programmers may save time and effort by not describing easier sub-goals that might have to be discarded, because programmers had to produce an alternative solution that is feasible.

- **Most-Critical-First.** The aim of this control method is to provide a strong skeleton around which the remaining sub-goals can be fleshed out.

- **Most-Interesting-First.** This approach can be used by the programmer to maintain his fading interest in the programming problem.

- **Mixture.** All the above control methods may be used in conjunction with varying priorities. For example a programmer may typically select sub-goals on the criteria of Easiest-First. However, there may be an alternative sub-goal that is sufficiently
critical and interesting to outweigh the easiest sub-goal, making the programmer choose the alternative.

1.5.4. Opportunism

The global control strategies described above prescribed that sub-goals are pursued in a structured fashion based on the relative positions in which the sub-goals appear in the goal hierarchy. For example, Children-First is a structured global control strategy that prescribes that after recognising a sub-goal that its siblings are recognised, and then its children are described. The order in which sub-goals are described is purely determined by relative positions, such as siblings and children. However, there may be control strategies that diverge from the principle that sub-goals are pursued according to their relative positions. For the purpose of this thesis, a control strategy that diverges from the structured approach is called Opportunistic, since the programmer selects in situ to pursue the most opportune sub-goal. Many factors may determine which is the most opportune sub-goal to pursue, including whether it is analogous to a solved sub-goal. Since there is potentially an infinite number of factors that may guide Opportunistic behaviour, it is easiest to describe Opportunistic control strategies as anything which is not guided purely by the structure of the goal hierarchy.

1.6. Rationale of Thesis

The aim of this thesis is to investigate the control strategies software engineers use to design programs by applying the methods and theories of cognitive psychology. The advantage of using cognitive psychology to the study of human reasoning is that the results from such studies can be used to provide a basis from which to improve software design tools that better complementing human reasoning. Software tools which are designed according to the cognitive processes of programmers have the advantage of reducing the requirements on the working (short-term) memory for particular tasks, thus enabling a programmer to perform more cognitive tasks, and to perform them with fewer errors. This thesis does not purpose or support a particular theory of working memory, but it accepts the general idea that there are limits to the capacity of the working memory. Cognitive cost/effort refers to the amount by which solving a problem occupies the working memory.

In order to produce well-designed software tools it is first necessary to examine the strengths and weaknesses of human reasoning when engineering new software. Chapters 4, 5, and 6 examine the control strategies employed by software engineers to design new software without tools to facilitate their reasoning. From the results, the natural strategies they employ can be determined. This provides information about the natural limits for reasoning about software.
Introduction

With this knowledge it is possible to design tools that support their natural strategies that will reduce the cognitive effort, and thus increase the limits of their reasoning. As a result a cognitive ergonomic software tool will improve the productivity of software engineers by increase the number of cognitive tasks they can perform in a given time, and by decrease the number of mistakes they produce per cognitive task. A successful attempt to apply the results gain in Chapters 4, 5, and 6 to the design of an editor was made. Chapter 7 describes an empirical evaluation of the editor.
Chapter 2

Psychological Literature on Software Engineering
Contents of Chapter

The Literature Review chapter introduces studies on the psychology of programming, and studies on other design domains that relate to programming. The chapter contains an overview and a critique of studies on programming knowledge, heuristic strategies and control strategies. Discussion of programming knowledge and heuristic strategies is concerned with their impact on control strategies and also the implication for experimental methodology used to study programming behaviour.

The chapter describes the schism on control strategies between studies that reported structured programming behaviour and studies that reported Opportunistic programming behaviour. The studies that reported structured programming behaviour observed programmers solving simple problems that do not characterise the problems met by programming in the 'real' world. The studies that reported Opportunistic programming behaviour may have mis-categorised some structured behaviour as being Opportunistic behaviour.

The chapter concludes that the aims of the thesis is best achieved by studying the choice of control strategies, the role that the environment plays in the programmer's choice of control strategies, and the effectiveness of the control strategies for producing well-designed software.
2. Psychological Literature on Software Engineering

2.1. Introduction

In Chapter I, problems facing the software industry were given as increasingly poor productivity and increasingly poor quality of produced software. An important component in the software industry is the software engineer who is responsible for the design of software, which determines the quality of the software, and affects the productivity of the software industry. By investigating the behaviour and performance of the software engineer, it is possible to determine what factors affect the design of software and consequentially the quality of the software. There exists a body of research that has investigated the behaviour and performance of designers and in particular software engineers. This chapter reviews this body of research and how it relates to the order in which software engineers design software.

In the following sections, the chapter will first discuss the relevance of design to software engineering. It will then provide a general description of the design process, and of the importance of the designer to the design process. By inference it follows that the software engineer is important to the design of software. Therefore, the chapter will then critique previous studies concerning the factors that influence the actions of software engineers. This chapter will also discuss why studying the order in which software engineers design software will particularly be of practical help to the software industry. In addition the chapter will describe how previous studies relate to the order in which software engineers constructed software.

2.2. Design

The activity of designing is the production an artefact that fulfils a set of constraints (Smith & Browne, 1993). Since software is an artefact (i.e. made by humans) that is supposed to behave in a prescribed manner (i.e. fulfils a set of constraints), it then follows that software is designed. Therefore general comments about the study of design are relevant to the study of software engineering.

In order to understand the success (or lack of it) of the design process, it is necessary to study the role of constraints and in particular the constraints that affect the behaviour and performance of the designer. Unlike general problem solving (non-design tasks) the constraints are often ill-defined and open to various interpretations (Simon, 1973). The sources of constraints are many and varied. For example, constraints include the requirements of the artefact’s purchaser, aesthetic considerations, the design environment, and limited resources, including finite cognitive resources of the designer. All these constraints have a bearing on the design
process and ultimately on the artefact itself. The constraints can therefore determine the success of the artefact.

Cognitive psychology can contribute to the goal of improving the effectiveness of design by investigating the factors that constrain the cognitive abilities of designers and how designers respond to these constraints. The results can be used to design tools that compensate for the cognitive constraints, and thus improve the probability and the degree of the artefact’s success.

2.2.1. Design as Problem Solving

A major concern of cognitive psychology has been how people think, especially how they think about problems. There is no universally accepted definition of what is a ‘problem’ within cognitive psychology. Reitman (1965, p.126) purposely provided a very general definition “a system has a problem when it has or has been given a description of something but does not yet have anything that satisfies the description”.

A system (e.g. a person) has a problem when it wants to transform the initial state (the description) into the target state (one that satisfies the description), where the transformation has to be formulated. For example how to solve ‘the tower of hanoi’ is a problem only until its solution has been formulated and remembered. Future attempts to solve ‘the tower of hanoi’ are not problematic because the solution does not need to be formulated.

2.2.1.1. Types of Problems

The above definition of what is a ‘problem’ encompasses many human activities from solving a mathematical equation, to building a house; from navigating, to explaining natural phenomena. Reitman (1965, pp. 131-142) described five different types of problems. His categorisation of problems was based on how well defined the initial and the target states are. Of interest to the study of design is Reitman’s second type of problem. That is where the initial state is ill-defined (or even not defined) and the target state is also ill-defined, which Simon (1973) calls ill-structured. Given that major components of any complex design activity involves discovering, understanding, and prioritisation of constraints (Machett, 1968), it seems that the second type of problem includes the activity of design. From Machett’s description of the design process it is clear that its initial state is not defined and that its target state is ill-defined as its constraints need to be ‘discovered’. Design is a problem that is ill-defined that results in the production of an artefact.

2.2.1.2. Information Processing

Cognitive psychology typically explains cognition as information processing. For example, people process information in order to transforms the initial state into the target state. “Information is something we get when some person or machine tells us
something we didn’t know before” (Garner 1962, p. 2), essentially a bit of information is something that provides certainty when it is known and provides uncertainty when it is not known. In the context of problem solving, humans process information in order to formulate how to transform the initial state into the target state. The uncertainty of how to formulate the transformation is resolved by processing information.

Processing of information involves acquiring information, and manipulating information to generate further information. There are many models of how people process information (Norman & Bobrow, 1976; Broadbent, 1984; Neisser, 1976), but they share some commonalties. A typical model would include Long-Term Memory, a Sensory Store, and Working Memory (also known as Short-Term Memory). People collect data about the world via their senses, and the sensory data is stored in the Sensory Store. Some sensory data is converted into information and placed into the Working Memory. The Working Memory uses information gained from the Sensory Store and from Long-Term Memory to generate new information, and the new information maybe transferred to the Long-Term Memory. Working Memory has a limited capacity (Miller, 1956; Norman & Bobrow, 1975), and it is where information is manipulated (Baddeley, 1989, p.36).
Many of the models see the external environment as a place that provides data to which people react. Neisser's (1976) model also sees the external environment as a place where people 'act' as well as 'react'. A consequence of this is that the external environment can be used by people. For example they can use the external environment as a place to store information.

Each individual model may (1) include additional components, (2) have slightly different attributes for each component, or (3) have slightly different relationships between the components. The merits of the individual models are not relevant to the studies described in Chapter 4-7, although the behaviour of designers is interpreted to be concerned with the processing of information. Furthermore the strategies they choose for processing information is influenced by the limited capacity of their working memory, and their ability to store and retrieve information in the external environment.

2.2.2. Design Expertise

As mentioned above, the designer needs fully define the constraints of the target state to solve the problem of design. Sources of constraint are many and varied; Lawson (1980) identified four main contributors to design constraints: the designers, the clients, the users, and the legislators. For example, the users require a system that is easy to use. The clients want a system that can use hardware that they possess. The designers want a database that reduces the repetition of data. The legislators require a system that complies with current laws (e.g. Britain's Data Protection Act 1984). During the design process the designers and other participants discover more constraints and gain fuller understanding of the constraints already known. The realisation of constraints thus determines both the nature and success of an artefact (Smith & Browne, 1993). How the contributors discover, understand, and prioritise constraints depends on many factors. For example the behaviour of designers many depend on their knowledge, skills, motivation, environment, or aesthetic tastes.

Expert designers are required to possess a range of different skills and knowledge to search for a solution path, which is not predetermined (Newell, 1969), for a complex (Rittel & Webber, 1972), ill-defined (Simon, 1973; Goel and Pirolli, 1992) and ill-structured (Reitman, 1965) problem known as design. Schraagen (1993) identified the following categories of design expertise:

- domain knowledge,
- heuristic strategies,
- control strategies.

**Domain knowledge** consists of the conceptual and factual knowledge associated with a particular domain. A programmer is required to have knowledge of computer science, programming languages, hardware and application domain (Guindon, 1990b). For example, the designer of a payroll system might require knowledge of pension
schemes, taxation, embedding SQL with C, network protocols, etc. Similarly, Brooks (1990) offers a typology of programming expertise ranging from knowledge that is specific to particular programming environments to knowledge that relates more to the application than to the software. It is fair to say that the majority of psychological research into programming expertise to date has focused upon the identification of domain knowledge. This has led most notably to the development of the 'plans' theory of programming expertise by Soloway and colleagues (e.g. Soloway & Ehrlich, 1984; Rist 1990).

**Heuristic strategies** are the tacitly acquired skills that specify how a design goal can be achieved, for example by selecting the appropriate domain knowledge or by decomposing the goal into a set of simpler sub-goals. A much favoured heuristic for the decomposing of goals is structured programming (Wirth, 1971; Parnas, 1972). Structured programming involves dividing a goal into simpler goals, and thence forming a hierarchy of design goals. The division of goals continues until simple structured building blocks are reached. A basic building block can be either sequential actions, repetitive actions, or conditionals. This is only a partial strategy, since it describes the constraints on dividing a goal, but it does not describe why one set of sub-goals are preferred over another set. This is as if saying “You are constrained to cut a cake into 4 equal sized pieces.” You still have to decide how you are going to slice the cake.

![Figure 2.2: Three Ways in which a Cake can be Divided into Four Equal Portions](image)

**Control strategies** are computational algorithms that specify the order in which design goals are to be achieved. As a component of design expertise, a control strategy is essentially a method by which designers prioritise functional and other constraints of a problem. A combination of heuristic and control strategies chosen by a designer determines the decomposition order of a design task and consequently influences the success of the finished artefact.

Whilst heuristic strategies such as structured programming constrains the division of goals, they do not dictate how one set of sub-goals that fulfils a constraint will be preferred over another set that also fulfils the constraint. For example, Ratcliffe and Siddiqi (1985) identified five principle ways in which the signals problem could be decomposed. Furthermore they outlined a set of factors, reflecting the design experience of the programmer, that determined the heuristic strategy a designer used to derive one of the five possible solutions. They did not, however, examine the control
strategies used to determine the order in which sub-problems were achieved. Since programs are a hierarchy of goals, the description of control strategies used in programming resembles the terminology used in Artificial Intelligence and Graph Theory to describe the traversal of hierarchies. Many researchers of programming prescribe that code should be constructed Top-Down and Breadth-First (e.g. Wirth, 1971; Dahl, Dijkstra & Hoare, 1972; McClure, 1975).

2.3. Domain Knowledge

In order to perform any cognitive task it is necessary to possess some form of knowledge representation. It is also necessary to be able to perform transformations/operations on the knowledge representations.

Software engineering is a cognitive task which involves a great deal of information that the engineer needs to represent. Brooks (1990) provided examples of the range of knowledge a programmer needs to possess. The list of examples started with knowledge highly specific to particular programming environments (domain knowledge) and finishing with knowledge about the application (application knowledge).

Researchers in the field of psychology of programming have studied some of this range of knowledge that engineers use in order to perform a programming task. Studies have either concentrated on the contents of knowledge, or the structure of knowledge. It should be noted that this divide is not clean, each has a bearing on the other. Some studies have also attempted to examine how these two aspects of knowledge alter with the expertise of the programmer.

The main area of research has been the structure of knowledge of a program (i.e. knowledge of the design artefact), and the structure of programming knowledge (i.e. knowledge used to guide the design process).

The precedence for this research comes from the notion of chunking data. Chase and Simon (1973) examined the performance of chess players to memorise the position of chess pieces. When the chess pieces were randomly positioned on the board, both the experts and the novices performed at similar levels. Thus, they argued that experts working memory resources are no greater than novices. However, when the pieces were meaningfully positioned on the board, the performance of the experts improved, while the performance of the novices remained the same. Indeed, experts could remember positions of more pieces than they should be able to if the experts remember each piece's position individually. In a classic study the number of discrete data a person can remember was discovered to be 7±2 (Miller, 1956). Therefore, the experts must possess some method of organising individual pieces into chunks (a formation), of which the novice is not aware. The ability of experts to process large amounts of meaningful knowledge has been found in a number of domains, including Go.
McKeithen et al. (1981) repeated Chase and Simon's (1973) Chess experiments in the field of programming. McKeithen et al. presented programmers, of various levels of expertise, an ALGOL program. Half of the programmers saw the ALGOL program scramble, with lines of code appearing in random (non-meaningful) order. The other programmers saw the ALGOL program with its line of code appearing in normal (meaningful) order. All programmers were asked to recall as much of the program as possible. Their results were similar to Chase et al. Experts were better than novices at remembering program statements arranged in a meaningful order. Both experts and novices performed poorly when asked to remember statements that were presented in a non-meaningful order.

However, not only can experts extract chunks of information from a representation, they can also make predictions about the whole representation when they have only viewed part of the representation. After extracting a partial chunk of information, they can predict the missing sections. People have expectations because they have experienced similar situations. This would suggest that people possess mental constructs specifying a chunk of prototypical actions and entities, which Schank and Abelson (1977) called scripts. Plans are scripts that specify a sequence of prototypical programming actions that achieve a programming problem.

2.3.1 Plans

A plan consists a series of slots, and a description of the goal it achieves. Slots can contain program actions, generic actions, links to other plans, etc. Although, the plan is generally flat, i.e. it is not hierarchical. The programmer chunks statements by examining the statements and trying to match them against plans.

Ehrlich & Soloway (1984) claims there are two main types of tactical plans to represent stereotypical actions in a program. These are Control Flow Plans, and Variable Plans (which can be considered to be Data Flow Plans). An example of a Variable Plan is the Running Total Loop Plan. This plan outlines the actions used to accumulate partial totals and the actions used to keep track of the number of numbers.

Ehrlich and Soloway's (1984) evidence of plans consists of presenting Pascal programmers of varying expertise with a program with one line blanked out. The line blanked out was a virtual action in a Variable Plan, such as initialisation of a variable. Experts were more able to fill in the blanks correctly than novices. Ehrlich and Soloway concluded that this was consistent with experts possessing plans that the novices did not possess. Ehrlich and Soloway recognised that this was very weak experiment, in that the results are consistent with many alternative hypotheses. The results show that expert programmers form knowledge that allows them to predict
Literature Review

The structure of knowledge has to be such that it allows for predictions to occur. The results do not provide evidence for preferring one knowledge structure (i.e. Plans) other another knowledge structure, where both enable predictions to be accurately made.

It is generally considered that Soloway's plans are language and task independent. The plans' actions are not coded as statements from a particular language, but experience with a particular language would lead the programmer to develop plans that are appropriate to that language, and possibly less appropriate to another language, but that nevertheless can be applied to other languages (Scholtz & Wiedenbeck, 1989). Rist (1986) termed Soloway's plans as 'Basic'.

In themselves Basic Plans do not provide enough information to complete programming tasks. There need to be (1) Plans that translate the program's goals/tasks to and from Basic Plans; and (2) Plans that translate the Basic Plans to and from program code. Rist termed (1) as 'Global' plans, while Soloway and Ehrlich (1984) termed (1) as 'strategic' plans and (2) as 'implementation' plans.

The question of whether plan knowledge is procedural or declarative has a bearing on the number of different types of plans. Procedural knowledge is use specific and is therefore unidirectional. In this case it is necessary to have one plan to translate program goals into Basic Plans and a different plan to translate Basic Plans into program goals. Declarative knowledge is not use specific and is therefore bidirectional. A declarative plan can be translated in either direction. So only one declarative plan is required where two procedural plans are required. This question will be examined later on.

The notion that Global and Basic plans are not encoded in a particular language's constructs entails that these plans are learnt while practising one language, and can be used by the programmer while using another language. Language dependent implementation plans on the other hand are not transferable. Scholtz and Wiedenbeck (1992) investigated experts and novices learning a new language. They hypothesised that the experts would perform better than the novices by making use of their plans. However they would use plans that provide the best solution in their original language, but not necessarily the best solution in the new language. This is precisely what they found to occur when Pascal programmers are learning to program in a language called Icon.

Robertson and Yu (1990) provided further evidence that the Basic plans are both language and task independent. Robertson & Yu asked programmers to identify coherent chunks of code within 9 different programs. For each program there was a Pascal version and a FORTRAN version. Pascal is a structured language, while FORTRAN is not. Subjects were only shown the version written in the language with which they were familiar. In both language groups the subjects identified chunks of
code that correspond to a plan. Programmers identified the same plans even if they where designing programs in different languages, and programs that performed different tasks.

Soloway (1984) mentioned that there were two types of Basic plans: Control Flow plans; and Variable plans. However most researchers have treated Basic plans and Variable plans as synonymous. This can be seen to have occurred in Pennington's (1987) research. Pennington outlined two types of knowledge organisation a programmer may have. The first she described is Text Structure Knowledge. Text Structure Knowledge outlines control flow structures. The second type she calls Plan Knowledge which is in fact Variable Plans. Plan Knowledge outlines data flow structures.

Pennington (1987) presented COBOL programs to professional COBOL programmers, and FORTRAN programs to professional FORTRAN programmers. Pennington asked the programmers to comprehend the programs. She then tested for priming effects and asked them questions about different aspects of the programmers. In the priming test, programmers were shown a line of code, then programmers were presented with a second line of code. The programmers had to respond whether the second line of code came from one of the programs that they had read. Some of the pairs appeared in the same control flow plan, and some pairs appeared in the same data flow plan. Pennington found that programmers responded quicker to pairs in the same control flow plan than from pairs in the same data flow plan. This would suggest that programmers organise their comprehension of the programs using control flow plans.

The comprehension questions also conformed to this conclusion. The type of plan that programmers use will determine which aspects of the program they will be able to answer. A programmer using control flow plans will be more able to answer question about control flow than a programmer using data flow plans. The converse is true for questions about data flow. The results showed that overall programmers used control flow plans. However the programmers who displayed a high level of comprehension appeared to have formed both a control flow and a data flow representation of the programs. Either the programmers who comprehend most formed both representations in parallel, or they formed the control flow representation first followed by the data flow representation.

In a second study, programmers were asked questions in the middle of a programming task, and at the end of the programming task. During the middle of a programming task, the programmers could answer a greater percentage of questions concerning control flow than questions concerning data flow. At the end of a programming task, programmers were able answer a greater proportion of data flow question than control flow questions. This indicates that programmers first formed the control flow representation, before they created a data flow representation.
Borrowing from Kintsch and van Dijk’s (1978) theory of natural language text comprehension (where abstract text representation is based on functional information of text), Pennington suggested that a data flow representation is dependent on a control flow representation. A control flow representation outlines the procedural information of the program, which defines the actions and events that need explaining. The data flow representation outlines the functional information, which explains the actions and events. Actions can not be explained until the actions have been defined. In short this means that programming goals are first decomposed into control flow goals, with data flow being dependent on the control flow. So in order to understand the order in which programmers decompose goals we need to focus on the order in which control flow units are produced.

### 2.3.1.1. Rist’s Model of Plans

An elaborate concept of programming plans has been proposed by Rist (1986; 1989; 1990; 1991). Rist hypothesised that there are three types of plans: ‘basic plans’, ‘complex plans’ and ‘goal plans’. The slots in a plan describe an arrangement of the plan ‘focus’ and supporting elements. The focus is the action that directly achieves the goal. For example within a plan that sorts a list of numbers, the focus of the plan is the swapping of elements (Wiedenbeck & Scholtz, 1989). The nature of the supporting elements is dependent on the plan’s type. For a basic plan the elements are simple actions that can normally be translated into a single line of code. The elements in complex plans are basic plans, and the elements in a goal plan are complex plans. The elements within the plans are arranged to identify the flow of data and control amongst the elements, which contradicts Pennington’s data.

The majority of studies investigating plans have concentrated on comprehension tasks. However, Rist (1989; 1990; 1991) attempted to investigate the predictions of his plan theory regarding the order in which code is generated. In essence a plan performs three operations: input (I); calculation (C); and output (O). Rist claimed that if programmers possessed the necessary plan to solve a problem, they will produce the solution in the order that it is executed (IC), whereas programmers without the plan will produce the solution by working backwards (CI). This hypothesis is based on studies that showed if a person does not known the solution to a physics problem he or she will work backwards from the goal state (Larkin, McDermott, Simon, & Simon, 1980). Rist’s studies confirmed that novices tend to work backwards, while more experienced programmers tend to work forwards.

Recently, Rist (1994; 1995; 1996) has described plans as a backward slice (Horwitz et al., 1989) from an object of interest, the plan’s focus. A backward slice shows flow data and control towards the plan’s focus. A plan is described in terms of a directed graph, where the root node of the graph is the focus of the plan. Each node in the
graph, including the plan's focus, represents either another plan or a line of code. Arcs in the graph are directed from one node \( x \) to another node \( y \) in the graph. An arc either represents backward data flow dependency from node \( x \), or backward control flow dependency from node \( x \). Rist (1994) developed Plan Analysis by Reverse Engineering (PARE) that could convert a Pascal program into a directed graph which represents the 'program plan'.

Once programmers have developed or selected a plan, they can determine from the plan four orthogonal bi-directional arcs emanating from a node to other nodes. So the plans describe a program in four independent dimensions. The first two orthogonal arcs/dimensions (Rist, 1995; 1996) are (1) control flow (indicating which node obeys and which one controls), and (2) data flow (indicating which node makes data and which one uses data). The last two orthogonal arcs/dimensions (Rist, 1996) are (3) linear order, and (4) class membership. The latter is only appropriate for object oriented programs.

\[\text{Figure 2.3: A Node with Linear, Control Flow, and Data Flow Arcs}\]

Rist (1995) has also an explanation of the cognitive processes, based on cue-search, that determines the order in which above plans are produced/selected. A description of cue-search is provided in section 2.5.1.3. In this thesis I am primarily interested in transitions from one node to another that move along the control flow dimension. Transitions that move along any of the other dimensions were called Opportunistic (in order to be consistent with Visser and Guindon). Opportunistic transitions are a secondary concern of this thesis.

**2.3.1.2. Procedural or Declarative Knowledge**

The form that programming knowledge takes has a bearing on the method by which that knowledge can be studied. Ryle (1949) noted that knowledge can either be knowing that (procedural) or knowing how (declarative). The way in which knowledge is encoded and therefore retrieved depends on whether the knowledge is procedural or declarative. Rumelhart and Norman (1985, p. 51) stated that "we seem to have conscious access to declarative knowledge; but we do not have this access to procedural knowledge." Thus, programmers will be able to report programming knowledge only if it is declarative. In order to employ a methodology that requires analysis of verbal
reports, it is first necessary to determine whether programming knowledge is declarative.

Pennington, Nicolich and Rahm (1995) conducted a study that determined whether programming knowledge is procedural or declarative. They started with ACT* (Adaptive Control of Thought - Anderson, 1983) that states knowledge acquired while learning a skill is encapsulated in procedures called production rules. Production rules are use specific which means that a rule can not be used for a sub-task for which it was not created. This means that transfer of skills is limited to cases where a sub-task of a target skill is the same as a sub-task of a source skill.

Pennington et al investigated the transfer of the programming skills ‘generation’ and ‘evaluation,’ from one to the other. They conducted a GOMS (Goals, Operators, Methods and Selection - Card, Moran, & Newell, 1980) analysis of the two skills to produce a set of production rules that each skill used, and then compared which rules the two skills shared. For this they were able to calculate the amount of transfer that will occur according to the ACT* theory.

The results contradicted the predictions that skills are encoded procedurally. Programmers transferring their skills performed far higher than expected in comparison to the performance of programmers who have the same level of experience of the new target task, no knowledge of the old source task, and initially had the same level of knowledge at the start of the study. The only difference between the two groups of programmers was that one had knowledge of the source skill while the other did not. So any difference in performance must be attributed to the fact that the knowledge of the source skill helped to achieve the target skill, i.e. there was transfer. Furthermore the greater the level of difference between the two groups indicates a greater level of transfer.

Pennington provided two explanations for this greater level of transfer than expected. Either the production rules produced by the GOMS analysis were not the same as those used by the programmers, or programmer’s knowledge is not procedural, but declarative. A protocol analysis indicated that the second alternative is the likely explanation. The declarative knowledge programmers possessed and could use to complete the generative task, and that they possessed and could use to complete the evaluation task displayed a high level of similarity, to the point of being nearly equivalent.

Since programming knowledge is declarative, programmers will be able to verbalise accurately their thoughts. Thus verbal protocol is a valid method for collecting data regarding programming knowledge.
2.3.1.3. Use of Plans outside of Pascal

A striking common factor amongst experiments that provide support for knowledge organised into plans is that they have concentrated on Pascal users, and mainly on comprehension tasks. Pascal was designed by Wirth (1971) a practitioner of structured programming. Bellamy and Gilmore (1990) have argued that plans represent structures within a program and that they are only useful when the program is structured, as are Pascal programs. They go even further by suggesting that when plans do occur they are ‘external’ structures rather than ‘internal’ structures. Just as an expert retrieves the appropriate knowledge demanded by the situation, the structure in which the knowledge is manifested is also dependent on the situation. The knowledge structure depends on the situation (e.g. the programming task, and the language).

Gilmore and Green (1987; 1988) attempted to improve code comprehension by highlighting the plan structure within the code. They tested this comprehension aid on Pascal and Basic, by asking non-expert programmers to discover errors within the code. At the time of the experiment Basic did not support structured programming, though it has since evolved into versions that do. The aid improved the performance of the Pascal programmers, while the aid had no effect on the behaviour of Basic programmers. This suggested that the Basic programmers do not use plans, or rather non-expert Basic programmers do not use plans. It may be easier for Pascal programmers to develop plans as they are more explicit in Pascal than it is for Basic programmers to develop plans. The results may not be repeated if expert programmers were used.

A prediction made by plans is that when a programmer is presented with a number of isomorphic problems, for which he possess a plan, the order of code generation will be the same for all of the isomorphs. Ormerod and Ball (1993) tested this theory on Prolog programmers, and Bellamy and Gilmore (1990) observed Basic, Pascal and Prolog programmers. From both studies there was no evidence that the programmers of any of the three languages constructed code using plans.

2.3.1.4. Summary of Plans

Much of the evidence for plans depends on the difference between the comprehension skills of Pascal experts and novices. The other source of support is the general order of Pascal code generation within plan structures. Yet, the evidence for plans is lacking when observing the order of code generation for isomorphic problems. However, plan use appears when FORTRAN programmers are explicitly told to chunk a piece of code, whereas plans are not apparent when implicitly required to comprehend Basic programs.

It would appear that plan use depends on the nature of the task and the programming language. However a more likely explanation of the observed behaviour
is the experience of the programmers. In each experiment the subjects in each group tended to have the same background. For example Robertson's Pascal subjects were computer science graduates from Rutgers University, New Jersey, and the FORTRAN subjects were engineering graduates from the same university. A difference between groups of subjects may be accounted by the training each group has received. Davies (1990) showed those software engineers who were given instruction on structured programming were more like to code in a structured fashion than less informed engineers.

As the existence of plans as an underlying knowledge structure is doubtful, it may be more useful to study programming behaviour by focusing on strategies rather than the structure or contents of the programming knowledge.

2.3.2. Analogies

The ability of designers to use analogies has consequences for the nature of domain knowledge. To enable the use of analogies, the domain knowledge needs to be able to specify similarities between different representations.

"If R is an analogical representation of T, then (a) there must be parts of R representing parts of T ... , and (b) it must be possibly to specify some sort of correspondence, possibly context-dependent, between properties or relations of parts of R and properties or relations of parts or T." — (Sloman, 1975)

Wharton et al (1994, 1997) argued that similarities between representations occur at several levels. The lowest and simplest level is element/object similarity where objects between the representations are similar. The next level is relational/situational similarity where relationships between objects are similar. The highest level is system/thematic similarity where goals and plans of the representations are similar. Analogies can share similarities at any or all levels. The similarities between different representation may be close or remote. The main difficulty for psychology is explaining how a representation reminds people of remote analogies, especially remote thematic analogies (e.g. the theme of the sour grapes fable and the theme of the anecdote about the disgruntled job seeker, Schank, 1982). Programming plans by themselves would not enable software engineers to be reminded of thematic analogies, nor remote analogies.

Several studies of programmers have observed programmers capitalising on analogies (Visser, 1990; Guindon, 1990a; Adelson & Soloway, 1984). Use of analogies is potentially very useful for software engineering (Jones, 1994), in that it enables software engineers to solve present problems by reusing solutions of previous problems. Most studies on analogies are concerned with the ability of people viewing one representation to be reminded of another representation (Wharton et al, 1994, 1997;
van der Meer, 1996). This is not an important question within software engineering. The important question is how software engineers recognise similarities between problem statements, and how they adapted a solution from one problem statement to solve a similar problem statement (Sen, 1997).

There are two potential methods by which a software engineer could adapt a solution from one problem to an analogous problem. To be able to reuse solutions it is first necessary for the designer to recognise similarities between two problems. The designer can then act in one of two ways (Détienne, 1991). (1) The designer recalls the past solution from his or her memory and adapts the past solution to solve the present problem (Guindon, 1990b). (2) Alternatively, the designer is unable to recall the past solution.
solution, but can recall where the past solution was used (Sutcliffe & Maiden, 1990; Neal, 1989). The designer then parses the solution where it is described, and adapts the solution to the new problem. Many additional stages of using analogies have been proposed, such as verification and justification of analogy (Kedar-Cabelli, 1988).

When designers use the first method their behaviour may conform with structured control strategies. Whereas, when designers use the second method their behaviour will diverge from structured control strategies and it will be characterised as being opportunistic (c.f. Section 2.5.).

2.4. Heuristic Strategies

2.4.1. Formal Heuristic Strategies for Decomposing Problems

The prime method for solving a problem is to ‘divide and conquer’ (Kant & Newell, 1984). Within the field of software engineering there are a number of paradigms for this heuristic. The predominate paradigm is structured programming. Structured programming involves constructing software using a small number of stereotypical code structures. A number of programming languages have been designed to support this approach. These include Pascal, C and ALGOL.

Another important approach is modular programming. A modular program consists of a group of code segments called modules. Each module can execute other modules. A module is constructed so that its internal information is hidden from other modules (Stroustrup, 1993). This means that modules can only affect other modules through the data they receive and give. As a result, the internal structure of individual modules can be altered without affecting the behaviour of other modules. Prolog is a programming language that supports modular programming, while Pascal is both a structured and a modular programming language.

Recently the paradigm of OOP (object-oriented programming) has received a high profile amongst software engineers. OOP is an elaboration on the modular programming technique. Essentially an object-oriented program consists of a set of objects. Each object has a number of attributes and a number of modules associated with it. A data structure is an instantiation of an object. The operations that can be performed on the data are restricted to the modules associated with its object type.

Each paradigm is not necessarily mutually exclusive. A number of articles have been written by software engineers outlining the advantages of their favoured technique. Wirth (1971) argued in favour of structured programming, Parnas (1972) supported modular programming, while Stroustrup (1993) thought that object-oriented programming was superior. The method by which programmers decompose a goal
affects the ecology of the recognised sub-goals, and this in turn may affect the best order in which to pursue the sub-goals.

2.4.2. Selection of Goal For Decomposition

The formal techniques, such as structured and object-oriented programming, place constraints on the way that goals are decomposed. However, a goal can be decomposed in a number of different ways, where each decomposition conforms to the constraints. The constraints merely limit the range of possibilities. The factors that lead a programmer to choose one goal decomposition over another must therefore be identified.

A major factor determining the behaviour and performance of programmers is the experience of the programmers. When programmers are faced with a known problem, or a problem similar to a known problem it is hypothesised that programmers will reproduce the solution they used previously to solve the problem.

Wu and Anderson (1993) investigated how programming experience of using different types of looping structures affects programmers' future solution to novel problems. A loop enables a section of code to be repeated until a condition is met. Within Pascal there are two such looping structures: while-do and repeat-until. The two structures differ on the position of the condition. In the while-do loop the condition is at the beginning, and is tested at the start of each loop. The repeat-until loop's condition is at the end, and is tested at the end of each loop. This difference has ramifications for when the use of each structure is appropriate. The appropriateness of a loop structure is dependent on the program's requirements.

Wu and Anderson created a number of similar problem statements. All the problems could be solved by using either while-do or repeat-until, but for some problems one structure was more appropriate than the other. One of their experiments was to present subjects with the group of problem statements that all favoured the use of the same looping structure. After the subjects had solved these problem statements, they were presented with problem statements that were neutral to the choice of structure. The results of the experiment showed that the structures selected for the neutral problems were the same as the structure used in the previous problems. So if the subject solved a number of problems using the while-do loop, they would solve neutral problems using the while-do loop. Previous experience with a similar problem appears to affect the choice of solution, and thus the decomposition of the goal, for the current problem.

Ratcliff and Siddiqi (1985) also investigated the effect of experience on the decomposition of a goal. They presented their subjects with the 'signals' problem statement. This problem in essence involves analysing a list of 1's and 2's, by counting the number of 1's, the number of 2's, and the longest sequence of consecutive
2's. There are two main solutions for the decomposition of this problem. The subjects were arranged into two groups. All the subjects had experience of structured programming. The second group had in addition received lectures that "stressed the need to consider the logical structure of data as a mean of obtaining an outline [solution]."

Each group differed by the decomposition that they adopted. The preferred solution within the second group reflected the structure of the data. The difference of behaviour was attributed to the difference of experience between the two groups. Unfortunately, the two groups differed on two factors. As well as receiving the extra instruction, the second group used Pascal while the first group used ALGOL. The two languages have similar properties, and as such it is unlikely this was the factor that produced the difference. However, this possibility cannot be ruled out.

Siddiqi and Ratcliff (1989) also considered the influence of the problem statement's wording on the solution produced by the programmers. It has been shown that the ease of solving a problem is dependent on the presentation of the problem (e.g. Wason & Green, 1984). Siddiqi and Ratcliff’s experiment also examined the consequence of the form of presentation, except that they were not examining the resulting ease of solution, but rather the resulting choice of solution. The subjects were presented with one of four problem statements. Siddiqi and Ratcliff managed to predict the decomposition that would be produced for each problem statement by counting the number of references to different abstract data items. For example two of the problems involve the processing of text, so the decomposition produced depended on the number of times the terms ‘character’, ‘word’ and ‘sentence’ were mentioned.

Despite the poverty of research on heuristic strategies, two factors that contribute to the decomposing of a problem have been identified. The first factor is the experience of the programmer, this includes practical experience with similar problems and theoretical knowledge of interpreting a problem. The second factor is the presentation of the problem. Like the possession of theoretical programming knowledge, this determines the way in which the problem is interpreted, and subsequently how it is decomposed.

2.5. Control Strategies

2.5.1. Global Control Strategies

While the heuristic strategies specify how to decompose a goal, it is the control strategies that specify the order in which the goals are decomposed.

2.5.1.1. Factors That Affect Control Strategies

When a software engineer chooses a control strategy to follow there are two main sources of consideration. The two sources are computational factors and psychological factors.
The computational factors are factors that determine whether the resulting design solution is a good one. A conscientious software engineer will want to select a control strategy that will enable him or her to produce a solution that fulfils the specified requirements. For example a good design solution might be one that does want it is meant to do and one that was produced with the specified time limits. Computational factors that ensure the solution is good, would include consistency between components of the solution.

The psychological factors are factors that determine whether the designer can employ a control strategy. It is foolish for a designer to select a control strategy that can theoretically meet the computational factors if he or she does not have the aptitude to follow the selected control strategy. For example people have working memory with limited capacity (c.f. Section 2.2.1.2.). Therefore a control strategy will fail if its cognitive costs (e.g. requirements to store memory) are greater than the available cognitive resources (e.g. spare capacity of working memory). Below is a more detail discussion of psychological factors that determine the success of a control strategy.

Psychological Factors.
The main factor determining use of control strategies that researchers have investigated is the knowledge possessed by programmers (Adelson & Soloway, 1985; Détienne & Soloway, 1989; Rist, 1986). For example, Rist’s model attempts to explain the order in which code is generated by the programmers’ knowledge of various mental ‘plans’. If programmers are faced with a goal to which they have a solution (plan) then they will retrieve the plan and implement it Top-Down, Breadth-First. However, if programmers lack knowledge of a relevant plan then they produce code Bottom-Up by concentrating on the goal’s focal activity and building around the focus.

Green, Bellamy and Parker (1987) extended Rist’s model by recognising the limitations of ‘internal’ working memory, and the importance of the ‘external’ world as an extension to memory. The extended model is called ‘parsing-gnirsp’\textsuperscript{10}. With a limited working memory it is necessary for programmers to use the external world as extension to their own memory. Whenever the programmers’ working memory is threatened with overload, they dump the code fragment with which they are working into the external memory. The dumped code is then parsed back into their working memory as the information is required.

As a result of working memory limitations, programmers can not design the whole program and then write the solution from start to end in a linear order. Programmers

\textsuperscript{10} The name refers to the two processes involved in programming: (1) the parsing of solutions; and (2) the creation of solutions. The creation of a solution can be seen as the reverse of parsing, i.e. gnirsp.
can only design fragments of programs (which correspond roughly to a plan) that need to be externalised before programmers can design addition fragments of the solutions.

This emphasis on 'external' features of programming led Green et al to recognise the importance of the programming languages and the editors' language. However, they were primarily concerned with how the notation of both languages affected the order of code production for individual complex plans as described by Rist, rather than with the order of code production for a program containing many complex plans.

Davies has examined some aspects of the parsing-gnisrap model, notably the importance of notation (Davies, 1991b), and the importance of externalising solutions (Davies, 1996). Davies (1991b) examined non-linear development of BASIC and Pascal programs by novice, intermediate, and expert programmers. Non-linear movements were categorised either as jumps within plans (as described by Rist), or as jumps between plans. If notation of a language affects the strategy of programmers then the number of between plan jumps and the number of within plan jumps should vary according to the language being used. Davies' results showed that the notation of the language does not affect the pattern of jumps for novices nor experts, but there is an effect for intermediate programmers. Davies concluded that novices are unable to capitalise on role-expressive notations, so that notational differences between program languages have no effect on the novice's strategies. On the other hand, experts can interpret obtrusive notation so that again the notational difference between languages has no effect on programming strategies. However, the intermediate programmer who is not fully competent with a language but who has learnt some aspects of the language will have a clearer understanding of highly expressive notations than obtrusive notations.

The main mechanism of the parsing-gnisrap model is working memory and its limitations. Davies (1996) also examined the idea that experts make better use of working memory than novices. In Davies' initial study experts and novice performed a programming task and a concurrent task that would reduce the capacity of the working memory. The performance of novices deteriorated when they performed the concurrent task. The concurrent task had no effect on the performance of the experts. In Davies' second study the use of external memory (i.e. the editor) was restricted, in that the programmers could not edit a line of code after they pressed return. As a result the experts' performances deteriorated beyond the performance of the novices, while the performance of novices was not the affected by the restriction of the editor. The two studies indicate that the expertise of programmers is not achieved by increasing the capacity of their working memory, but by gaining strategies that rely on the use of the external memory.

Davies (1991b; 1996) has shown that parsing-gnisrap is correct in identifying the external environment as a source of memory. However since experts have spare
working memory capacity, Green et al (1987) were incorrect in identifying the limited capacity of working memory as the reason why programmers use external memory. In addition, the notation of a language can affect the performance of intermediate programmers, but the performance of experts is not significantly affected by the obtrusiveness of notation.

2.5.1.2. Structured Control Strategies for Designing Software

There have been a number of studies investigating the order in which a design solution is produced for complex problems. These studies fall broadly into two categories. The first category is where researchers have observed a structured approach to the construction of the solution's hierarchy. Within a structured control strategy the designer describes each node that he has recognised in a systematic order. Jeffries, Turner, Polson and Atwood (1981) recognised two different methods of Top-Down design. Novice programmers produced a solution using a Depth-First strategy. Anderson and colleagues (Anderson, Farrell & Sauers, 1984; Pirolli & Anderson, 1985; Pirolli, 1986) have also observed novices using a Depth-First control strategy. This is where the programmer pursues the completion of a goal as soon as that goal has been recognised. Experts used a Breadth-First method. Within a Breadth-First method, a programmer attempts to describe the goals in the order that they have been recognised.

The structured approach has received further support from Adelson and Soloway (1984; 1985) and Rist (1990). In Adelson’s experiment, expert software designers were observed maintaining a strict Breadth-First strategy. Even when the designers discovered an opportunity they resisted capitalising immediately on the discovery if it evolved diverging from the current level of investigation. Rist also noted Breadth-First behaviour amongst experts. He had also observed that novices work Bottom-Up, by focusing on a salient sub-problem, and then by designing around it.

Ball and Ormerod (1995) expanded on the notion that experts use Breadth-First while novices use Depth-First. They argued that designers used a mixture of the two approaches depending on the design context. While designers are solving a novel problem they attempt to maintain a Breadth-First strategy since this approach facilitates the design of a robust solution. Unfortunately, Breadth-First is cognitively demanding. The demands of a sub-problem and the strategy may be overwhelming, forcing the designer to adopt the simpler Depth-First strategy. Another instance where experts may
use Depth-First is when they face a problem to which they know the solution. The designer does not have to design the solution but merely retrieve and implement it.

2.5.1.3. **Cue-Based Search**

The structured control strategy's description of the order in which programs are designed does not explain the cognitive processes that directs the control strategy. In an attempt to address this shortcoming, the parsing-gnisrap model of Green *et al* (1986) extended the information encapsulated within plans to include the precondition for using a plan, and the post-condition after using a plan. When programmers are developing plans, they first select the 'focal lines\(^{11}\). The programmer then tries to match the precondition of the focal lines with the post-condition of basic plans. If a match is found then the matched basic plan is added to the program plan, and a new precondition is produced to continue the process of creating a program plan.

Rist (1995) made a similar alteration with his model of planning by introducing to the psychology of programming the notion of cue-based search. The principle behind cue-based search is that memories are indexed by cues, whereas the method of solving a problem is to generate cues and then searching various memory sources for a memory whose index cue matches the search cue. The result of the search is then used to generate new search cues. This continues until all the cues have been matched, at which point the problem has been completed.

Rist (1995) describes a cue as a tuple of three keys: role, goal and object\(^{12}\). When the designer searches for information, only one key of a cue needs to have a value, although more than one key may be defined for any search. If one of the keys of a cue is not defined, then the return from a search only provides partial information. For example, a goal cue (where only the goal of a cue is defined) can only return a goal structure. For a final solution the role, goal and object structures need to be known, and combined. This can be achieved either by searching with a complete cue, or by searching with a series of incomplete cues and combining their returns. Incomplete cues are more likely to find matches than complete cues are.

For programmers to engage in a cue-base search, a cue is not enough. They also need a search command. The command defines which way (backward or forward; up or down; role, goal, object, or plan; data flow, control flow; or linear flow) to search,

---

\(^{11}\) The focal lines are the lines of code that achieve a particular goal, but which require supplementary code to support it. For example, the code that swaps two adjacent numbers is the focal lines for the goal of sorting numbers.

\(^{12}\) Roles related to procedures as found in procedural design. Goals related to functions as found in functional design. Objects relate to objects as found in object oriented design.
and what to do with the return of a search (i.e. code or make a note). The search command specifies a design strategy which synonymous with the control strategy.

At the end of each search, the programmers can either change the search command or the search cue. Rist (1995) explains that programmers typically maintain the same search command but alter the search cue. The new search cue is produced using the return of the previous search. This chain of searches (known as an episode) continues until the return of the last search can not be used to generate a new search cue. At this point the programmer must find a new search cue, and possibly change the search command. This continues until there are no remaining cues that have not been linked to a node. If all cues have been linked, then the problem has been solved.

Rist (1995) has used the cue-based search model to explain the behaviour of a student programmer observed during a previous study (Rist, 1989). The cue-based search model provides a strong explanation of any control strategy used by programmers. As a result of the model’s ability to explain any control strategy, the predictive power of the model is weak, in that it cannot predict which control strategy will be used.

2.5.1.4. Opportunistic Behaviour

Although many studies have observed programmers handling goals in an orderly fashion, there has been some research (Guindon, 1990a; Visser, 1990) which suggests that programmers employ Opportunistic Control Strategies which are less structured than Top-Down Strategies. A common characteristic of studies that have observed structured behaviour is that subjects were given ‘toy’ problems. When programmers were given more realistic tasks (i.e. ill-defined, c.f. Section 2.2.2.) they started to perform Opportunistically.

What the term ‘Opportunistic’ implies is rather hazy. Guindon (1990a) views Opportunism as a data-driven control strategy, whereas structured strategies are goal-driven. This is based on Anderson’s (1983) view of experts possessing procedural knowledge encoded as production rules whose antecedent is activated by the presence of particular data. For Guindon a partial solution to a goal acts as data, and an expert programmer will use such data to guide them in expanding on the partial solution. For Guindon, Opportunistic actions are actions that have not been planned from the outset of implementing the solution, while Guindon characterised the structured Top-Down approach as being planned from the outset.

Visser (1990) has a different concept of ‘Opportunism’. For her, a programmer has many factors influencing the choice of next action. One factor is the programmer’s goal
agenda, which she calls a plan (not to be mistaken for Rist's plans\textsuperscript{13}). As a result of the many factors, a number of actions are proposed. The programmer selects the action that is cognitively cost-effective. For Visser all actions are Opportunistic, the programmer selects the most opportune action that has been proposed. It is possible to view the actions suggested by the goal agenda as being goal-driven, while all other suggested actions are data-driven.

*Planning, Opportunism, and Situated Actions.*

Opportunism (Hayes-Roth & Hayes-Roth, 1979) and Situated Actions (Suchman, 1987) are both rebuttals to the notion that planning prescribed actions will enable people to achieve their goals. The rebuttal of opportunism and situated actions follows from the idea that people operate in an uncertain world, and therefore plans are bound to fail because they cannot take into account all factors that will impinge on achieving a goal (Guindon, 1990a; Winograd & Flores, 1986; Norman, 1993).

Opportunism and Situated Actions differed on whether mental representations can cope with uncertainty. Theories of Situated Actions rejected the importance of mental representations while they emphasised the importance of the situation (i.e. relationships between an individual, others and the environment) for guiding behaviour (Suchman, 1993). Opportunism maintained that mental representations are important for guiding behaviour, but that they are continually being changed to reflect the changing situation (Vera & Simon, 1993).

Opportunism believes that plans are generated to be followed, but that they are subject to change (Visser, 1990) and temporary distractions (Newell, 1987). Vera & Simon (1993) noted that “most plans are not specifications of fixed sequences of actions, but are strategies that determine each successive action as a function of current information about the situation,” i.e. there may be a plan of *actions*, but the execution of the next immediate action is determined by a plan of *action* that is dependent on the situation.

Theories of situated actions deny that even the next immediate action is determined by a plan (Suchman, 1993). Within theories of situated actions it is unclear where computation of the next action is performed, principally because devotees of situation actions are not interested in the question of the computation’s location, instead they are interested in relationships that guide behaviour. It seems that they think that behaviour is guided by unconscious reaction to the situation, while plans are consciously constructed prospectively and/or retrospectively to make sense of unconscious

---

\textsuperscript{13} Rist's plans are mental constructs that outline generic solutions to common problems. Whereas, Visser's plans are goal agendas outlining how to implement a specific solution to a problem.
reactions. The dichotomy between Opportunism and Situated Actions maybe that Situated Actions see plans as being conscious constructs, while Opportunism does not. So the unconscious action of determining the next immediate action is described as planning by theories of opportunism, but not as planning by theories of situated actions.

**Opportunistic Planning**

Descriptions of opportunistic planning have been based on blackboard models (Visser, 1990). Blackboard models consist of two components (1) a knowledge source that typical contain production rules, and (2) a blackboard data structure. The knowledge source is analogous to the long-term memory, and the blackboard is analogous to the working memory. Each production rule describes actions to perform if a given situation exists. The content of the blackboard represents information of the current situation. The contents of the blackboard are matched against the antecedent of the production rules. There maybe several production rules whose antecedents match descriptions of the current situation. A selection criterion is applied to the matched production rules, to choose one production rule to follow. The actions specified by the chosen production rule are then performed. Then the blackboard is updated, and the next production rule is chosen, etc. (Nii, 1986). There have been many proposed methods of how to choose the production rule to follow, and on when the selection is made (Anderson, 1993, p. 93).

Plans of actions can be encoded within the knowledge source, and given the situation maybe the source of the selected action. The action specified by the plan is not always the one that is selected. Visser (1990) proposed several reasons why divergences from a plan of actions may occur. For example, plans break-down when a situation occurs that the plan of actions has not considered. In such a break-down situation divergence from the plan occurs, and the plan is only resumed when the break-down has been resolved.

**Opportunism as Divergence from Structured Control Strategies**

It is possible that in some cases the proposed goal-driven (planned) action and the proposed data-driven (opportunistic) actions are the same. This makes it difficult to determine a single cause for an action, i.e. is it planned or is it opportunistic. Within this thesis any transition from one node to another node that is follows a structured (i.e. control-flow) relationship is considered to conform to a structured control strategy. Any transition between two nodes that is determined by any other factor is considered to be Opportunistic. In short anything that does not conform to a structured control strategy is characterised as being Opportunistic.

It is curious that opportunistic studies on program have assumed that Top-Down behaviour must be planned (Guindon, 1990a; Visser 1990). In addition, Ball &
Ormerod (1995) have argued forcefully that the use of the term Opportunistic is a misapplication, however in order to remain consistent with the literature (i.e. Guindon, 1990a; Visser 1990) the thesis will term any action not following a Top-Down structured control strategy as being Opportunistic.

Both Visser's (1990) and Guindon's (1990a) models assume that more actions will diverge from a structured strategy than will conform to a structured strategy. To test their theories they measured the number of structured divergent and structured convergent actions. So if an action does not conform to structured Top-Down behaviour then it is an indication of Opportunistic behaviour. If a large count of structured divergent actions is observed then it is difficult to deny the existence of Opportunism.

Guindon (1990a) reported that at least 53% of her subjects' movement from one action to another diverged from structured behaviour. Visser did not attribute a percentage to pure divergent behaviour, but it is clear from the data that her subject never completed a goal in a single continuous Depth-First sequence.

Problems with Studies on Opportunism

However both studies are characterised by a similar problem, a weak definition of what constitutes structured performance. Any activity transition that does not conform to either a Depth-First strategy or a Breadth-First strategy cannot automatically be considered to be divergent. Visser asked her subject what control strategy he intended to employ, he replied Depth-First. Subsequently, Visser labelled any activity transfer that was not Depth-First as divergent, despite the possibility that the subject decided to work Breadth-First.

Further examination of Visser's (1990) paper reveals that, even neglecting the possibility of Breadth-First, the number of divergent activity transitions was only 24%. She reports 43 transitions from an operation in one sub-goal to an operation in another sub-goal (1990, Figure 6), and a total number of 181 transitions from one operation to another operation (1990, Table 5).

An additional problem of Visser's (1990) results is the possibility that a transition from an incomplete sub-goal to another sub-goal occurred when the subject thought that he had completed the problem. In such a circumstance the subject can be actively conforming to a structured approach, but which Visser considers to be divergent. This explanation seems plausible when one notices that much of a sub-problem's solution is completed in one continuous sequence.

As mentioned, Guindon's (1990a) experiment also had a problematic definition of structure congruent performance and structure divergent performance. Guindon produced a list of rules that specified as structure congruent (balanced), any activity transfer that is Top-Down. She also produced an additional set of rules that outlined
Opportunistic performance (unbalanced). There is a conflict between the two sets of rules (Ball & Ormerod, 1995). Any Depth-First behaviour (drifting) under Guindon’s rules can be categorised as either structured or Opportunistic.

Guindon (1990a) chose to view Depth-First behaviour as Opportunistic. A third of the observed divergent behaviour was Depth-First (Ball & Ormerod, 1995). If the Depth-First behaviour is re-categorised as structured, then the 53% figure is reduced to 35%. Guindon’s claim that the majority of programmers’ activity transfer is divergent is cast into doubt.

As a result of incorrectly classifying structured behaviour as divergent, it appears that Guindon (1990a) and Visser (1990) observed less Opportunistic behaviour than they had thought, but it appears they did observe some Opportunistic behaviour.

2.5.2. Local Control Strategies

Within Visser’s (1990) model a number of possible actions can be suggested. It is necessary for the programmer to resolve which action to perform. A similar problem occurs for software engineers employing a Breadth-First control strategy. After describing a sub-goal, the programmer has to select the next sub-goal to describe from one of the sub-goal’s peers.

Visser proposed that the selection of action was based on two criteria: Cognitive cost; and importance. The first criterion is more influential than the latter. The cognitive cost amounts to the ease of completing an action. The cost of action is cheap if the programmer has performed the action before and already knows how to execute the action. The availability of information also determines the cognitive cost. To execute an action it is necessary for supporting information to be at hand. If the necessary information is not known, the cost of the action is high since cognitive resources have to be directed to the discovering of the information. Visser also expanded on the importance criterion. The importance of the actions depends on the importance of the type of action. For example, actions that fix omissions for a previously described goal are considered to be important. The importance of the goal affected by the action determines the importance of the action.

Ball and Ormerod (1995) suggested that cost-effectiveness determines the next goal to be described. The cost-effectiveness of a proposed goal is a determined by (1) the resulting likelihood that the finished artefact conforms to the specifications, (2) the goal’s cognitive cost, and (3) the overall cognitive cost of completing all remaining goals after completing the proposed goal. Visser’s cognitive cost leads to selection of goals according to their ease of completion, whilst Ball and Ormerod’s cost-effectiveness leads to the selection of goals according to proven design practices (i.e. ensuring that a good solution is produced).
2.5.3. Summary of Control Strategies

The research into control strategies used by expert programmers has revealed diametrically contrasting results. When programmers solve 'toy' problems they appeared to apply a structured approach. However for more realistic complex problems, programmers are reported to be less structured (Guindon, 1990a; Visser, 1990). Unfortunately within the latter studies the possibility that the programmers were using a structured strategy that was not prescribed by the researcher had not been considered.

2.6. Focus of Research

The thesis has outlined three main areas of software design covered by studies in the psychology of programming: Domain Knowledge; Heuristic Strategies; and Control Strategies. The field of domain knowledge has concentrated on the structure of knowledge, and in particular the notion of plans. The studies that have investigated the reality of plans have provide problematic results. Plans have typically only appeared during comprehension tasks, and mainly involving the use of Pascal. Bellamy and Gilmore (1990) have argued that “knowledge is fluid and its characteristics, as revealed by experimental data, are determined more by the characteristics of the experiment than by the conceptual organisation.” This suggests it would be more profitable to study the use of knowledge in a particular task context rather than study the organisation of knowledge per se.

To this aim the thesis is concerned with investigating the effects of the environmental context on the control strategies engineers use to construct software artefacts. In other words, what are the factors in the ‘external’ world that determine which strategy a programmer selects. How are the programmers constrained by the ‘external world’, and how do they respond to the constraints? There are many constraints in the outside world, these range from the programming culture, to the users’ requirements, to the programming tools that are available. The thesis plans to concentrate on the effects of the programming tools on the choice of control strategies, as this area offers the best scope for improving the performance of the software industry. A programming tool is any artefact that programmers use to design and implement software, this includes programming languages, and editors. Chapters 4, 5 and 6 investigate the effect that programming languages have on the choice of local and global control strategies, while Chapter 7 investigates the effect editors have on the choice of global control strategies, and the advantages of the Breadth-First control strategy.
Chapter 3
Protocol Analysis
Contents of Chapter

The chapter on protocol analysis discusses the information required to analyze the behaviour of the programmers and the various sources of data (verbal and keystroke reports) that will be collected in the following experiments to provide the required information. The chapter then describes the strengths and weaknesses of the data from the various sources of information, and how to collect the data from the various sources. In particular the chapter details how a goal hierarchy is constructed from the program's text based on the building blocks of the programming languages. The program's text is constructed in turn from the verbal reports and the keystroke reports. The final section of the chapter outlines how the analysis of the verbal reports and the analysis of the keystroke reports can be combined to provide a description of programming behaviour that is consistent with both the verbal and keystroke reports.
3. Protocol Analysis

3.1. Introduction

The aim of the following studies is to understand the order in which programmers construct a solution, and the reason for constructing the solution in one particular order rather than in another order. The purpose of the data collected is to provide evidence for answers to these questions.

The evidence needs to take a form that outlines a series of events, where the attributes of an event consist of an action, a place of action, and a time of action. The result of the protocol analysis is to produce a series of triplets, such that each triplet describes an event that contributes to the construction of a solution to a programming problem.

3.1.1. Type of Action

Generalisations need to be made in order to be able to analysis the data collected. Each event consists of an unique action that occurs in an unique situation. By generalising the action and by extension the situation in which it occurred, it becomes possible to translate the roughly described action observed in one situation to a rough hypothesised action of another situation.

Previous attempts have been made to generalise actions involved in designing and implementing a program. The only justification for a particular scheme of generalisation, is how accurately the scheme describes a series of events, and how accurately the scheme can be used to hypothesise about a future series of events. If an event occurs which the scheme can not generalise then that scheme has failed with respect to that event.

A scheme for generalising actions which has had some success is one developed by Ormerod and Ball (1993) from previous schemes such as Ericsson and Simon (1984) and Fisher (1987). A couple of minor alterations to Ormerod and Ball’s scheme were made to account for actions that might be made in the tasks presented to programmers in the following chapters, but which would not have been made in the tasks presented by Ormerod and Ball. For example, in the following studies participants were able to search for information from many more sources than participants could in Ormerod and Ball’s study.

The modified scheme is displayed in Appendix B.1. The scheme allows an action to be generalised into one of eight categories. Each category is made up of a number of sub-categories. This enables each action to be generalised at two levels. The most general level described the purpose of the action, such as 'Searching for Information'. The second levels provided additional details. For example, the general action
'Searching for Information' can be described more specifically to contain detail about which information source is being searched, e.g. 'Reading Code', 'Questioning Environment', etc.

3.1.2. Place of Action

Since for this research, the propose of employing protocol analysis is to discover the order in which programmers construct a program, the place where an action occurs should be described as a location within a program. Enumerating location within a program relies on the fact that programs have a hierarchical structure. A hierarchy consists of a set of nodes, and a programming event takes place at one of these nodes. By giving each node a unique index number (identity) it is possible to indicate the position where an event occurred.

A requirement of indexing is to discover the characteristics of each entity (in this case nodes) and identify the key characteristics, i.e. the characteristics which are unique to each entity. In situation where a key characteristic does not exist, a combination of characteristics may be used, provided that the combination of the chosen characteristics is unique to each entity.

Every node in a program can be uniquely identified by the combination of its parent's identity, and the order of its execution in comparison to its siblings\(^{14}\). In the following studies siblings can not be executed in parallel. So the index of a node is a combination of the index of its parent and the order of execution.

Therefore a node is given an index of X:n, where (1) it has a parent with the index of X, and (2) it is executed after n-1 of its siblings have been executed. The period in X:n is merely a separator and not the multiplication operator. The top node is given the index of 1, as it has no parents and it has no siblings to be executed before it is.

![Figure 3.1: Example of Enumerated Nodes](image)

\(^{14}\) This method of enumerating nodes does not hold for parallel programs, however within this research parallel programs have not been investigated.
3.1.3. **Time of Action**

In the analysis of the verbal and keystroke protocols reported in this thesis, the exact time at which an action occurs is not important. However, the order in which the actions occurred is important. Order of actions can be described by several means. For example, the order of actions can be described by referential links to preceding actions, and referential links to following actions. An alternative method of describing order of occurrence is to record the time of occurrence for each action. The order of occurrence can then be determined by comparing time of occurrence of each event.

The second method was chosen as it encodes more information than the first method (although the additional information may not be required). An additional reason is that the first method contains repeated data, which makes it more difficult to maintain consistency within the data collected. The base time, time zero, is when the computer is switched on.

3.1.4. **Source of Data**

An important requirement for collecting data is to identify the possible sources of data and to identify the scope and limitation of the various sources. Two sources of data are the verbal reports of thoughts and actions, and recordings of interaction between the programmer and the machine environment. The first source produces verbal protocols, while the second source produces keystroke protocols.

By collecting data from more than one source, it is possible for the limitations of one source to be overcome by an alternative source. Furthermore, different sources of data can be used to verify each other, where the scope of the different sources overlap.

Both verbal data and keystroke data provide an incomplete description of the events that occur during the construction of a solution. However, the incompleteness of each source occurs for different events. So the two sources of data complement each other.

The keystroke data only provides information on the interaction between the programmer and the machine environment. Such recorded interaction is concerned solely with writing down (or deletion) of solution components. The keystroke data can not provide any information on purely mental events such as evaluating the solution, and the scheduling of goals. However, keystroke data provides complete and unambiguous information on events that involve interaction between the programmer and the machine.

Verbal data provide information on all types of events involved in designing and implementing a solution. However, for various reasons programmers do not or can not verbalise all of their thoughts. So while verbal data provides information on all types of events, it does not provide information on all events. The interpretation of verbal data is also problematic, since verbal reports can be ambiguous as their meaning relies on context, whereas the interpretation of keystrokes does not.
By analysing both sources of data, and combining the results it is possible to offset partially the limitations of each source. The verbal reports provide a rich source of data that cover all types of events, whilst the keystroke data provide complete and accurate data on the events involving writing the solution down.

3.2. **Verbal Protocol Analysis**

3.2.1. **Justification of Collecting Verbal Protocols**

Collecting verbal reports involves the participant performing a secondary task of concurrent verbalisation. The problem with performing a secondary task is that it may affect the participant's performance and behaviour in the primary task (indeed, on occasions that is its purpose). It is not always advisable to introduce a secondary task that affects the primary task, if the aim of a study is to understand how participants naturally achieve the primary task, as is the case in the following studies.

If a secondary task is to be included it is necessary to understand its potential affects on the primary task, and whether this invalidates the data collected. This problem of using verbal reports was considered by Ericsson and Simon (1980). They proposed three levels of verbalisation tasks, and they described how verbalisation tasks at the various levels may affect performance of the primary task. The level of a verbalisation task is dependent on the number of intermediate processes that occur between thinking a thought and verbalising the thought. Level one verbalisation tasks involve no intermediate processes. Level one verbalisation is reporting information content of the short term memory that is already encoded in a verbal form. Level two verbalisation tasks involve the recoding of non-verbal thoughts into verbal thoughts. Higher levels, level three and above, include processes that modify information contain, as well as modifying the form of information.

Ericsson and Simon (1980) hypothesised and demonstrated that level one verbalisation has no effect on the primary task. Level two verbalisation tasks may cause the participant to require more time to complete the primary task, and the participant may abandon verbalisation when cognitive resources become scarce.

The design of the following studies operates at level one, given that the primary task is language based, and that the verbalisation task used does not require modifying information content. Even if the assumption that the participants' thoughts are in a form that is ready verbalised is incorrect, the participants should never operated at a higher level than two as they asked to report all thoughts without modifying them. On the basis of Ericsson and Simon's (1980) account, the verbal reports collected in the following studies should not affect the thoughts produced by the participants. It is possible that producing the verbal reports may slow the participants down, but as the
analysis of data is not concerned with duration of completing a task, this is an acceptable side effect.

3.2.2. Collection of Verbal Data

The verbal data are collected by encouraging the participants to think aloud, and recording the utterances on audio tape. The only difficulty with collecting verbal data is to persuade the participants to continue talking. Whenever participants become silent for longer than 30 seconds they are prompted to talk.

3.2.3. Treatment of Verbal Protocols

The verbal data collected will be subjected to a number of processes. These are (1) transcription; (2) coding; (3) validation; and (4) analysis. Validating the coding of the verbal data is achieved by comparing it with the coding of the keystroke data. This process is described in Section 3.5. The analysis of the coded data is dependent on the question being asked. This means that the analysis process varies for each study, and as such the description of the analysis process is provided in the method section of Chapters 4, 5, 6 and 7. Each stage of processing the verbal protocols was performed by myself while knowing the author of each verbal report. This potentially caused the problem that the reports were misinterpreted. To offset this problem a second person, not knowing the author of each verbal report, checked the results of the verbal protocols.

The transcription of verbal data involves writing down every statement, word for word, that the participants had uttered. The other aspects of verbal data such as intonation, and length of pause will not be included in the transcription, except when such aspects contribute to the meaning of an utterance. The transcription produced will also included time-markers at one minute intervals.

The coding of verbal data first involves segmenting the data into utterances. An utterance starts at the beginning of a sentence or at the beginning of a conjunctive (e.g. and, or, but). An utterance finishes at the end of a sentence or at the beginning of a conjunctive. When the utterances have been identified, the purpose represented by each utterance will be examined. The general assumption is made that each utterance represents only one cognitive action being undertaken by the speaker. When the cognitive action represented by an utterance has been identified, the action is represented by a code (as outlined in Appendix B.1.). If the action is concerned with a particular section of the goal hierarchy, then this information of location is also coded. At the end of the coding process a list of events will have been produced.

3.3. Keystroke Protocol Analysis

The second form of data that will be collected in the following studies is the interaction between the participants and software tools (in particular the text editors used). The
principal form of the interaction was the text entered into the text editors by the use of a keyboard. The keystrokes also include the use of short cuts for the selection of functions belonging to the software tools.

In addition to keystrokes, two other forms of interaction between the participant and the software environment were recorded. A second form of interaction is the participant's use of the mouse, which includes the selection of the software tool's function via menus; the selection of text; and the manipulation of windows.

The third form of interaction is those initiated by the software tools which included the communication of problems via visual cues (dialogue boxes) and/or audio cues (a short sharp beep).

### 3.3.1. Collection of Keystrokes

#### 3.3.1.1. Logger-2

The initial method used to collect keystrokes was a tool called Logger-2. Logger-1 was a tool developed by Ormerod and Ball (1993) that extended the capability of the software environment LPA MacProlog 2.5. MacProlog is an environment for developing PROLOG programs on Macintosh computers. Logger-1 enabled a user to capture the contents of MacProlog and save the information (i.e. the program), with the time of capture also being saved into a log file. To initiate the capture of content with Logger-1, the user has to select the function from a menu using a mouse. Logger-2 enabled the capture function to be activated using a keystroke short-cut. As keyboards can be chained, the participant and the experimenter can use different keyboards. So the experimenter can select the capture function without disturbing the participant.

During the first study the experimenter captured contents of MacProlog at significant stages of implementing a software solution. This method of collecting keystroke protocols provided a rough guide of the action performed by the users, and a rough guide of when the user performed certain actions. The roughness of this method of collecting keystroke data was compensated for by the parallel collection of verbal data.

#### 3.3.1.2. TextRec

In the studies reported in Chapter 5, 6 and 7, a software tool TextRec (developed and implemented by the author) and variation of that tool was used to collect accurate data of interaction between the user and the computer. Logger-2 was an extension of a commercial software editor. Because of the difficulties (with current technology) of altering the functionality of compiled software, it was felt that the best means of collecting accurate data on interaction was to develop a software editor from scratch with the required data collection facilities included.
TextRec was developed to mimic Apple's editor TeachText with the added functionality of recording information about each instance of interaction and saving the information in a log file. As far as the users of TextRec were concerned it was identical to TeachText. The only perceivable difference was that TextRec beeped every minute, the propose of which was to provide a time stamp for the verbal data which made it easier to combine the verbal data with the keystroke data.

3.3.2. Data Collected

The data collected by TextRec is saved in a data file that is organised as a database. The file consists of a series of records. Each record relates to a single interaction between the user and the computer, and each interaction is tabulated into a field. Every interaction record consisted of three compulsory fields, and a number of additional fields dependent on the type of interaction being recorded.

The three compulsory fields are (1) the time when the interaction occurred, (2) the type of interaction, and (3) the source of interaction. The time of interaction is measured in the number of ticks (60 ticks = 1 second). The base time is when the computer was switch on, but as the tick time when the task started was recorded it is possible to calculate the time when an interaction occurs relative to the start time of the task.

The field 'type of interaction' stores the general nature of an interaction with more specific data being stored in other fields. For example if an user of TextRec changes the insertion point of text, the ‘type of interaction’ field will record that the user has changed the point of insertion, while an additional field will record the location of the new point of insertion.

The third compulsory field ‘source of interaction’ records how the interaction was instigated. There are three methods of instigating an interaction. An interaction can either be instigated by (1) TextRec, or by the user either via (2) use of the keyboard or via (3) use of the mouse. The use of the mouse is sub-divided into two parts: (3a) interaction with a menu; and (3b) interaction with a window. This sub-division is necessary because some actions, such as closing a window, can be achieved by using the mouse to interact either with a menu or with a window.

Appendix B.2. outlines the form of data recorded in a log file, with particular emphasis on the additional fields that are dependent on the type of interaction.

3.3.3. Coding of Keystroke Data

The keystroke data of each programmer is coded by first constructing a goal hierarchy of the solution from the final solution produced by the programmer. Each keystroke record is examined in the order they are produced. The propose of the examination is to
Protocol Analysis

identify when, where and how the keystroke contributes to the final state of the solution.

For certain interactions the question of where the action contributes to the final solution is not appropriate. For example, the interaction record relating to the start of the task does not relate to the construction of any one section of the solution. At this stage of the coding it is not possible to answer the question of how non-localised protocols contribute to the final solution. The 'how' question can only be answered once the verbal and the keystroke data are examined in parallel.

As a result of the difficulty of answering question about non-localised keystroke protocols at this stage of analysis, the analysis concentrates on finding when, where and how the remaining keystroke records contribute to the final solution.

When a keystroke occurred is easily identified by the recorded time of event in a field of each record.

Where in the goal hierarchy a keystroke contributes to development of the solution is determined by the location of the insertion point in the program text and knowledge of with which node that section of text is associated. The location of the insertion point is determined by previous keystrokes. It is therefore necessary to examine the protocols in order of occurrence, so that the location of insertion point can be calculated.

As the goal hierarchy is initially constructed from the final state of the solution, there are instances were a keystroke refers to event on a particular piece of text that does not appear in the final solution, and therefore does not appear in the goal hierarchy. Consequently it is necessary to revise continually the goal hierarchy by including the nodes associated with a piece of text that has been later deleted.

How a keystroke contributes to the final solution is determined by the type of action being undertaken.

3.4. Construction of Goal Hierarchy from Text of Software Solution

As mentioned in the previous section it is necessary to have a provisional goal hierarchy in order to interpret the action of the programmer as reported by the keystroke data. This section reports how a provisional goal hierarchy is constructed from the final text of the software solution, and how the text of the solution at intermediate stages of the problem solving activity is used to amend the provisional goal hierarchy.

The provisional goal hierarchy is constructed from the program text by identifying building blocks and their relationship to each other. This is achieved by using the structure of the solution's programming language to comprehend the structure of the solution. Each language has its own particular structure that specifies the construction of a valid building block. Type two and lower languages, such as computer.
programming languages and human languages, by definition allow a building block to be made up of instructions (sentence) and building blocks (of similar or different types). For example, a conditional building block may in turn be made up of smaller conditional building blocks. It should be noted that the analysis is not concerned with the grammar of the programming language, but rather the relationship of each instruction (sentence) to each other.

A program is itself a building block made up of smaller building blocks. Breaking down a program into its components creates a hierarchy. As each building block represents a goal, the hierarchy created is a goal hierarchy.

To illustrate this analysis a generic programming language is described below using Backus-Naur Form (Cohen, 1986). BNF can describe hierarchical structures in a form of production rules. The rules are written as \( X ::= Y \), meaning \( X \) is made up of \( Y \), where \( X \) is a symbol and \( Y \) is a string of symbols. A symbol can either be terminal or non-terminal. Non-terminal symbols are identified by being enclosed in angle brackets, and must appear on the left hand side of at least one production rule. \( I \) means 'or', so \( X ::= Y \mid Z \) means \( X \) can be made up of \( Y \) or made up of \( Z \). \( + \) means 'one or more' so \( X ::= Y^+ \) means \( X \) can be made up of \( Y, YY, YYY, etc. \)

### BNF 1: Rules Describing Generic Relationship of Building Blocks.

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;module&gt;</code> ::= <code>&lt;block&gt;</code>+</td>
</tr>
<tr>
<td><code>&lt;block&gt;</code> ::= <code>instruction</code></td>
</tr>
<tr>
<td><code>&lt;loop&gt;</code> ::= <code>test</code> <code>&lt;block&gt;</code></td>
</tr>
<tr>
<td><code>&lt;condition&gt;</code> ::= <code>test</code> <code>&lt;block&gt;</code></td>
</tr>
</tbody>
</table>

Each instruction is a token representing one of these building blocks. For example the following program
So Figure 3.2 shows the hierarchical goal structure of the above small program.
The goal hierarchy is created by interpreting the program text with knowledge of which building block each instruction represents, and how the spatial relationship of each instruction in the program text specifies the relationship between the building blocks and the instructions.

However, since the final program text does not include the text generated and later deleted, it is necessary to view the solution at each stage of construction. Whenever some text is discovered that does not appear in the final program, then the goal hierarchy is amended to include the structures represented by the additional text.

After producing a provisional goal hierarchy from the final program text, and amending it to account for the program text at all other stages of completion it is possible to interpret where and how each keystroke contributes to the generation of the goal hierarchy.

3.4.1. Ambiguity

There is a source of ambiguity in the relationship between the building blocks. The cause of the ambiguity is possibility that \texttt{<block>} X may be made up of Y (a number of \texttt{<block>}'s) which is in turn made up of Z (a series of symbols). Y, the intermediate \texttt{<block>}'s, is not represented by a token in the program text. The presence of Y may be indicated by comments or by the white spaces, but this is not necessarily the case. Where the presence of Y is not indicated, it is possible to interpret that X is made up of Z, without the intermediate Y.

During this stage of analysis it is assumed that intermediate levels are not present, unless comments and white spaces indicate otherwise. This is one reason why the goal hierarchy generated from the program text is treated as provisional.
3.5. Combining Verbal and Keystroke Protocols

Both the verbal protocols and the keystroke protocols are used to produce their own provisional goal hierarchies. Both protocols provide incomplete information of the structure of the goal hierarchy and the order of designing the goal hierarchy. Verbal protocols are not complete because programmers for various reasons fail to verbalise all their goals and sub-goals, and the interpretations of the verbal protocols may be incorrect because the verbal reports can be ambiguous. On the other hand, the keystroke protocols can be incomplete because not all goals are represented by tokens in the program text, and whenever a programmer considers a sub-goal he/she does not necessarily interact with the editor because the sub-goal has already been implemented or because the programmer is not convinced that the sub-goal is constructive to the goal.

The combining of verbal and keystroke protocols involved two tasks. The first task is the creation of a complete goal hierarchy that contains all sub-goals that have been considered and whose structure is consistent with both verbal and keystroke protocols. The second task is to determine the order in which the goals are pursued.

3.5.1. Combining Goal Hierarchies

The main source of inconsistency between the verbal provisional goal hierarchy and the keystroke provisional hierarchy is that they are created from incomplete protocols.

3.5.1.1. Missing/Extra Goals

As a result, the major source of difference between the two provisional goal hierarchies is that one hierarchy may contain a node which the other one does not. For example, the verbal protocol goal hierarchy may have an intermediate goal between main and the goals display flow and display wait (c.f. Figure 3.3), while the keystroke protocol goal hierarchy may not contain the intermediate goal, but contains the extra goal display average (c.f. Figure 3.4).

![Figure 3.3: Verbal Protocol Goal Hierarchy](image-url)
As a general rule any goal mentioned in either of the provisional goal hierarchies is accepted. However, where the protocol is interpreted as referring to a goal which was not reported by the protocol of the other form, the protocol is re-examined to determine whether the interpretation of the goal’s existence is correct.

There are a few occasions were it is not immediately apparent where the extra goal should appear in the final hierarchy. In the above example, the extra goal display average could either be a child of main or a child of display results. In such circumstances the purpose of each potential parent goal is examined to determine whether it is compatible with the sub-goal’s purpose. If only one of the potential parents is compatible with the sub-goal, then that potential parent is considered to be the actual parent. If both potential parent goals are compatible with the sub-goal, then the least abstract potential parent goal (the lower level goal in the hierarchy) is considered to be the actual parent. In the above example display average is compatible sub-goal of both main and display results. As display results is the least abstract of the two, it is considered to be the parent. Following these guidelines, the combined hierarchy produced from the two provisional hierarchies is shown in Figure 3.5.

3.5.1.2. Different Arrangement of Goals

Although the most common source of difference between the two provisional goal hierarchies is missing nodes, the most problematic difference is when a goal (or especially a set of goals) appears in both hierarchies but in different positions. The most common cause of this difference is a misinterpretation of one set of protocols by the analyst.

The misinterpretation is most likely to occur with the verbal data rather than the keystroke data. The goal hierarchy of the keystroke protocol is drawn up by the
examination of the program text. The examination is guided by 'context-free' rules. This means that the hierarchy was determined by syntactical relationship of components, whilst the verbal data are interpreted by the semantics of each utterance. Semantics is context dependent thus involves a greater amount of interpretation than syntax. Hence semantics provides a greater scope for misinterpretation than syntax.

As there is a more opportunity for misinterpretation to occur during the verbal data analysis than the keystroke data analysis, the greater reliance was placed on the results of the keystroke data analysis (i.e. the arrangement of goals as determined by the keystroke data was assumed to be more likely the correct arrangement).

The verbal data that were interpreted as outlining an alternative structure were re-examined to determine whether they can be interpreted as supporting for the structure outlined by the keystroke data analysis. If re-interpreting the verbal data to fit the interpretation of the keystroke protocol is not possible, then both sets of data are examined closely in conjunction with each other to uncover an explanation of the programmer's behaviour that is mutual acceptable to both sets of data.

3.5.1.3. Altering the Description of the Goal's Location

The analysis of each data produced a description of the location where the goals were pursued. The description involved giving each node an identity. However, the identity is based on one of the provisional hierarchies. When the final hierarchy has been produced it is necessary to amend the identities of nodes to correspond to the final position of the nodes.

3.5.2. Order of Goal Pursued

When the final goal hierarchy has been produced it is possible to determine the order in which the nodes of the hierarchy have been considered. The analysis of each data set provides a provisional order in which the goals have been considered. Each analysis also provides a time at which a node has been considered.

Producing the final order of goal pursued involved interleaving the two provisional lists on the basis of time each goal is pursued. Part of the process of interleaving the two lists is to identify identical items in both lists making sure such items are not duplicated in the final list.

The final list is an ordered set of triplets. The triplet contains three fields: (1) the node considered, described by its identity; (2) the time of consideration; and (3) the nature of consideration (e.g. generating, evaluating, selecting node is describe, etc.). The triplets are order by time of consideration. Some of the triplets are not concerned with one particular node, and for these triplets a null value is assigned to the field 'node considered'.
Chapter 4

The Global Control Strategies of Expert Prolog Programmers
Contents of Chapter

The chapter describes the conflict of results between those who have observed structured control strategies and those who have observed Opportunistic control strategies. Both sets of previous experiments have contained weaknesses. The chapter describes an experiment to examine the global control strategies used by expert Prolog programmers. The experiment's methodology and analysis is designed to overcome the weaknesses of the previous experiments. The aim of the experiment is to determine whether Prolog programmers use a Structured or an Opportunistic approach to design. The chapter also provides an *a priori* description of the advantages of the various structured control strategies.

The results of the experiment indicate that programmers predominately use a combination of the Children-First and the Depth-First global control strategies. The data was examined to determine when programmers used the Children-First control strategy and when they used the Depth-First control strategy. The analysis shows that programmers use the Children-First control strategy except when programmers are tackling an easy sub-problem or when the sub-problem relationship with other sub-problems is a disjunctive, in which case they used the Depth-First control strategy.

The data collected is also examined to understand the reason why programmers occasionally diverged from a structured control strategy. Eight reasons for divergent behaviour was discovered. The frequency of each reason causing divergent behaviour was measured.

The chapter concludes with a discussion of why programmers used a mixture of Children-First and Depth-First control strategies, rather than the Breadth-First control strategy which is supposed to produce the best results. It is probable that the Breadth-First control strategy is too cognitively expensive for programmers to use.
4. The Global Control Strategies of Expert Prolog Programmers

4.1. Introduction

As mentioned in Chapter 2, the literature review, studies are divided on the strategies programmers use to control the design of software. Many experiments in the early 1980s provided evidence that expert programmers designed software using the Breadth-First control strategy (e.g. Adelson and Soloway, 1985; 1986; and Jeffries et al., 1981). However, the studies which observed the use of structured control strategies have been criticised (Guindon 1990a; Visser 1990) for using simplistic tasks that lacked many of the characteristics of 'real' design tasks. When more realistic tasks were used (e.g. Ratcliff & Siddiqi, 1985; Ullman, Stauffer & Diettrich, 1986; Guindon 1990a; Visser 1990) the results indicated that the order in which expert designers tackle task components may be influenced by Opportunistic behaviour rather than by structured behaviour. The Opportunistic studies have in turn been criticised for being very restrictive in their definition of structured control strategies, for example Visser (1990) did not characterise Depth-First activity as being structured, while Guindon (1990a) did not characterise Breadth-First activity as being structured. Ball & Ormerod (1995) argued that programmers use a mixture of Breadth-First and Depth-First control strategies, consequentially both Guindon's and Visser's experiments over counted divergence from structured control strategies.

In order to resolve uncertainties surrounding the control strategies employed by expert designers and to assess properly the compatibility of design aids with design expertise, further empirical observations of complex design activities are required, using more detailed activity analyses than have previously been presented in the literature.

In this Chapter I will report a study on expert Prolog programmers producing solutions to the 'Signals' problem (described in Appendix C.3.), a complex task used by many researchers to explore programming expertise (e.g. Ratcliff & Siddiqi, 1985; Green, Bellamy & Parker, 1987; Ormerod & Ball, 1993). Data are provided which show that experts adopt a predominately structured rather than Opportunistic approach to decomposing design problems. However, I will argue that the structured approach they adopt is not one of the generally recognised pure Breadth-First or Depth-First approaches to Top-Down problem decomposition. Instead, expert Prolog programmers appear to adopt what I have termed a 'Children-First' approach to problem decomposition, in which the relative advantages of Breadth-First and Depth-First approaches are maximised whilst the disadvantages of these approaches are minimised.
Before reporting the observational study, I will first discuss the potential advantages of different control strategies that experts may use to decompose design problems.

4.2. A Priori Advantages for Each Control Strategy

In developing hypotheses about the use of different control strategies in expert design, it is important to distinguish between computational\textsuperscript{15}, psychological\textsuperscript{16} and pragmatic\textsuperscript{17} advantages that each strategy may confer. In terms of computational advantages, a Breadth-First strategy is generally recognised as prescriptively optimal design practice (e.g. Wirth, 1971; Dahl, Dijkstra & Hoare, 1972) for producing the best solution. In most design tasks, there are occasions when a goal in one section of the hierarchy has a bearing on the description of a goal in another part of the hierarchy. As the goal stack in Table A1 illustrates, in adopting a Breadth-First strategy, the completion of any goal is deferred until all goals in the hierarchy have been recognised. Thus, it supports a minimal commitment mode to conceptual design (Goel and Pirolli, 1989; 1992). A Breadth-First strategy is the only control strategy that can reliably identify interactions between distant goals prior to completion of any goal.

In purely computational terms, a Depth-First strategy is clearly sub-optimal, since low-level goals are completed prior to the recognition of higher level goals in other parts of the hierarchy. The programmers using a Depth-First strategy is committed to an early instantiation of a sub-goal’s solution and may not identify its inadequacies until it has been completed and they have started to solve another sub-goal. However, in psychological terms, adopting a Depth-First strategy has the advantage of placing the smallest cognitive cost upon the designer (c.f. 7.1.1. Parsing required for each global control strategy). Goals that have been recognised and described been must be maintained prior to their completion, either in the programmer’s memory or in some form of external repository. In the example goal decomposition shown in Table A1, a Breadth-First strategy requires the programmer to maintain a stack of recognised goals containing up to eight items. Adopting a Depth-First strategy gives a maximum of four goals in the stack at any one time (as shown in Table A2).

The relative importance of computational and psychological factors in the choice of control strategy is complicated by the influence of pragmatic considerations, relating to the management of realistic design tasks. For example, Ball and Ormerod (1995) argue that adopting a Depth-First strategy allows the designer to confirm the adequacy of their proposals for a particular sub-goal. This may be particularly important in realistic

\textsuperscript{15} Computational advantages are concern with the design solution.

\textsuperscript{16} Psychological advantages are concern with the individual’s cognitive requirements.

\textsuperscript{17} Pragmatic advantages are concern with miscellaneous advantages (i.e. anything which are not computational or psychological), such as management issues.
design contexts, where designers must provide evidence to a project manager of their solution’s viability. On the other hand, adopting a Breadth-First strategy also confers some pragmatic advantage, in that it enables a project manager to assess the scale of the emerging design solution prior to the commitment of further resources.

So far, I have considered only the relative a priori merits of Breadth-First and Depth-First strategies. Ball and Ormerod (1995) argue that, because of the computational, psychological and pragmatic factors outlined above, it is not practicable for expert designers to adopt a purely Breadth-First or Depth-First approach. Instead, they argue that it is more realistic to expect designers to switch between these strategies as appropriate. I will suggest that the Children-First strategy outlined in the previous section presents an alternative structured approach which combines to some degree the advantages of both Breadth-First and Depth-First approaches, yet avoids the ambiguity and structure-deviation of mixing these control strategies.

For individual designers, the main advantage of adopting a Children-First strategy is that, like adopting a Depth-First strategy, their attention is fixed on only one subsection of the goal hierarchy. During Breadth-First design the programmer is continuously viewing different sections of the hierarchy. Since it is impossible for the designer to remember the entire hierarchy (except for the simplest problems), the designer using a Breadth-First strategy must continuously parse information required to understand the current goal, but which may be irrelevant for the next goal. In the example decomposition, a Children-First strategy requires a maximum of four goals in the stack at any one time (as shown in Table A.3). To maintain this stack requires fewer cognitive resources than a Breadth-First strategy, and so to some extent a Children-First strategy has the same psychological advantage as the Depth-First strategy.

However, a Children-First strategy maintains to some degree the computational advantages of a Breadth-First strategy. It is true that the adoption of a Children-First strategy leads to the completion of some goals before recognition of the full goal stack is complete, though completion occurs later than with a Depth-First strategy. More significant, however, is that the description of high level sub-goals occurs much earlier with a Children-First than a Depth-First strategy. In other words, adopting a Children-First strategy enables the full exploration of the overall design hierarchy at an earlier stage than a Depth-First strategy. Thus, a Children-First strategy can be

---

18 In the example, each parent only has two children, if the children to parent ratio was greater, then the maximum number of goals in the stack would have increased for both the breadth-first and children-first control strategy, but not for the depth-first control strategy. If the depth of the example hierarchy was increased, then the maximum number of goals in the to be described stack would have increased for all structured control strategies.
Global Control Strategies of Prolog Programmers

classified as an intermediate method between the extremes of Breadth-First and Depth-First design.

A Children-First strategy also confers unique pragmatic advantages on the design process. When large software projects are undertaken by a team of developers, the goal of the project is divided into a number of sub-goals, and the boundaries of each sub-goal are defined by outlining their pre-conditions and the post-conditions. When the boundaries have been agreed, each sub-goal is developed by a sub-division of the team, although the evaluation of the sub-goal’s solution may be undertaken by other team members (e.g. Curtis, Krasner, & Iscoe, 1988). The Children-First strategy offers an appropriate model for the management of team design, since it offers the focus required by each development group on a particular sub-problem, whilst the early description of the goal hierarchy enables a managerial overview of the project to be maintained.

4.3. The Observational Study

4.3.1. Rationale

Empirical research to date suggests that, when programmers solve ‘toy’ problems, they appeared to apply a structured approach, but for more realistic complex problems programmers are reported to be less structured. Three important issues remain to be resolved:

- First, the control strategy used by expert programmers is unclear. It may be, for example, that expert programmers use a Children-First control strategy, an approach not previously recognised by other researchers since it does not conform to their definition of structured control strategies. Alternatively, experts may use a pure Breadth-First or Depth-First approach or a mixture of these modes. Finally, it may be that expert program design does not conform to any of these structured approaches, but is more Opportunistic in nature.
- Second, the conditions under which experts might switch between different global control strategies have been determined theoretically (Ball and Ormerod, 1995), but have yet to be observed empirically.
- Third, it is yet to be demonstrated empirically whether short term notions of cognitive cost and importance, or longer term notions of cost-effective design practice motivate local decisions concerning goal prioritisation.

An observational study was carried out of the control strategies used by experienced Prolog program designers, using keystrokes, written notes and verbal protocols as data. To determine the control strategy used by designers, the nature of the transitions from one sub-goal to another that were evident in the protocols are examined. Each strategy allows for some transitions and prohibits others. By comparing the observed transitions between goals with those allowed by the various strategies it is possible to

- 68 -
determine which strategy or combination of strategies provides the most complete account of the designers' behaviour. Also, by studying the verbalisations, the types of structure-divergent behaviours and local goal prioritisation decisions can be identified.

4.3.2. Definition of Control Strategies

Operational definitions are given in Table 4.1 for Breadth-First, Depth-First and Children-First strategies: these were used in coding activity transitions under each strategy. Each transition from one node in the goal hierarchy to another node was identified as conforming to or diverging from a strategy. The operational definition of a conforming transition depends on two factors: (1) the family relationship between the previous node and the new node; and (2) whether alternative confirming nodes had already been coded. Note that each transition can conform to a number of strategies. If an activity transition did not conform to a strategy it was marked as structure-divergent.

Table 4.1: Functional Definitions of Activity Transition Conforming to a Structured Approach

<table>
<thead>
<tr>
<th><strong>Breadth-First Global Control Strategy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Moving from the current node to one of its siblings</td>
</tr>
<tr>
<td>2 Moving from the current node to a node of the same generation, if all of the current node's siblings have been described</td>
</tr>
<tr>
<td>3 Moving from the current node to a node of the next generation, if all nodes of the current node's generation have been described</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Depth-First Global Control Strategy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Moving from the current node to one of its children</td>
</tr>
<tr>
<td>2 Moving from the current node to one of its siblings, if the current node does not have any children</td>
</tr>
<tr>
<td>3 Moving from the current node to a sibling of one of its ancestors, if the current node does not have any children</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Children-First Global Control Strategy</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Moving from the current node to one of its siblings</td>
</tr>
<tr>
<td>Moving from the current node to one of its children, or one of its sibling's children, if all the current node's siblings have been described</td>
</tr>
<tr>
<td>2 Moving from the current node to a child of one of its ancestor's siblings, if all of the current node's siblings and children, and all of the current node's siblings' children have been described</td>
</tr>
</tbody>
</table>

- 69 -
Global Control Strategies of Prolog Programmers

The definitions in Table 4.1 for each of the three control strategies are very strict. This ensures that the measurements obtained for conforming to a strategy will be conservative. After partially describing a goal, it is possible for the designers to believe that they have described the whole goal and consequently shifts their attention to the next goal. So even if the designer is actively applying a control strategy in such a circumstance, the transitions would be marked as divergent. Therefore, the above definitions for conforming to a strategy are likely to result in a score that is below the true score of conformance. To overcome this problem, the definitions were weakened in a secondary analysis to obtain a more liberal score. Where it was stipulated in the first analysis that "all of the current node's sub-goals have been recognised", in the second analysis this was changed to "half of the current node's sub-goals have been recognised". A number of transitions marked as structured-divergent in the conservative definitions might therefore be marked structured-convergent in the liberal definitions. Reclassified transitions were subsequently analysed qualitatively to determine whether they were divergent or convergent.

Although a high ratio of structure-divergent to structure-conforming transitions is often recognised as being indicative of Opportunistic behaviour (e.g. Guindon, 1990; Visser, 1990, Davies, 1991a), Ball & Ormerod (1995) argue that Opportunistic behaviour can consist of transitions that are structure-conforming, and that structure-divergent activities are not necessarily Opportunistic in origin. Therefore, a secondary analysis of verbalisations associated with structure divergent activities makes it possible to categorise the types of structure divergence.

4.4. Method
4.4.1. Participants
Four experienced Prolog programmers were observed in the study. The first (A1) had 18 years experience of designing and implementing programs in a variety of languages. He had used Prolog for ten years. The second (A2) was a PhD student who had used Prolog for two years within the commercial sector, plus a further year at university. The third (A3) had also worked within the commercial sector prior to registering for a PhD. He had 13 years programming experience, during which time he used Prolog for 10 years. The fourth (A4) worked in industry for ten years, and she had used Prolog for six years. The protocols of A1, A3 and A4 were fully analysed. A2's protocols were analysed up to the point when an error was discovered in the input data with

---

19 The liberal and conservative definitions of a depth-first strategy are identical since the designer's perception of whether a goal has been described does not affect whether his/her transition conforms to, or diverges from, the strategy.
which he had been provided. An additional programmer, AS, observed, but as he was unable to solve any sub-problem it was decided he did not have sufficient expertise to fulfil the requirements of a participant for this experiment.

4.4.2. Task and Materials

The designers were given a description of the requirements that their solutions had to fulfil (shown in Appendix C.). The design problem was configured to ensure that the solution's hierarchy was deep. Within shallow hierarchies a Breadth-First strategy and a Children-First strategy make the same predication on the order in which goals are recognised and described. To compare the two strategies it is necessary for the hierarchy to have at least four levels. The problem was configured so that the participants would be likely to produce a solution exceeding the minimum requirement of four levels. The programming environment was LPA MacProlog 2.5. All the designers had experience with this environment. A data recording program written in Prolog and available from the authors was run concurrently to record and time-stamp the emerging solution.

4.4.3. Procedure

Each participant was instructed by the experimenter to verbalise everything he or she was thinking during the experiment. The participant was then given a short programming task to familiarise them with thinking-aloud (shown in Appendix C.2.). After completing the practice task, they were given the main task (shown in Appendix C.3.). They were informed that they could use paper to jot down ideas and that the supervisor would answer any questions regarding the input/output file predicates, since these predicates were non-standard. During the main task, verbal protocols were recorded and the emerging solution was logged after each burst of keystroke activity. Any piece of paper used by the participants was also collected as data. The participants were given a total of two hours to complete both the practice task and the main task.

4.5. Results and Discussion

4.5.1. Global Control Strategies

Table 4.2 shows the percentage of activity transitions conforming to the conservative definition of a control strategy or a combination of control strategies. It is worth noting initially that there is a considerable overlap of agreement between Breadth, Depth and Children-First control strategies. The main source of this overlap is movement between siblings. Any movement between siblings is consistent with both Breadth-First and Children-First control strategies, also in at least 50% of cases such movements also conform to a Depth-First control strategy, when each branching node has at least two children (i.e. the majority of sub-goals are at the bottom of the hierarchy).
Global Control Strategies of Prolog Programmers

Table 4.2: Conservative Measure of Moves Conforming to Structured Control Strategies

<table>
<thead>
<tr>
<th></th>
<th>Breadth</th>
<th>Children</th>
<th>Depth</th>
<th>Breadth &amp; Children</th>
<th>Breadth &amp; Depth</th>
<th>Children &amp; Depth</th>
<th>All Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>42.29%</td>
<td>44.00%</td>
<td>72.57%</td>
<td>45.14%</td>
<td>77.14%</td>
<td>77.14%</td>
<td>78.29%</td>
</tr>
<tr>
<td>A2</td>
<td>50.56%</td>
<td>53.37%</td>
<td>69.66%</td>
<td>53.93%</td>
<td>81.46%</td>
<td>82.02%</td>
<td>82.58%</td>
</tr>
<tr>
<td>A3</td>
<td>47.57%</td>
<td>48.54%</td>
<td>69.90%</td>
<td>51.46%</td>
<td>80.58%</td>
<td>78.64%</td>
<td>81.55%</td>
</tr>
<tr>
<td>A4</td>
<td>47.78%</td>
<td>48.89%</td>
<td>60.56%</td>
<td>50.56%</td>
<td>73.89%</td>
<td>73.33%</td>
<td>75.00%</td>
</tr>
<tr>
<td>X</td>
<td>47.05%</td>
<td>48.70%</td>
<td>68.17%</td>
<td>50.27%</td>
<td>78.27%</td>
<td>77.79%</td>
<td>79.36%</td>
</tr>
</tbody>
</table>

From Table 4.2 it appears that the best fit with the observed activity transitions is the combination of either Breadth-First and Depth-First, or Children-First and Depth-First control strategies. Presupposing programmers use all three strategies merely complicates the model without significantly increasing the number of transitions that are accounted for. Table 4.3 shows the results of applying the weaker definitions of control strategies and then systematically resolving differences between the liberal and conservative definitions. Again, a combination of either Breadth-First and Depth-First, or Children-First and Depth-First control strategies, accounts for the most transitions. Interestingly, applying the weaker definitions increases the maximum number of transitions accounted for by only approximately 4%.

Table 4.3: Combined Conservative and Liberal Measure of Moves Conforming to Structured Control Strategies

<table>
<thead>
<tr>
<th></th>
<th>Breadth</th>
<th>Children</th>
<th>Depth</th>
<th>Breadth &amp; Children</th>
<th>Breadth &amp; Depth</th>
<th>Children &amp; Depth</th>
<th>All Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>47.43%</td>
<td>50.85%</td>
<td>72.57%</td>
<td>53.14%</td>
<td>80.57%</td>
<td>80.57%</td>
<td>82.86%</td>
</tr>
<tr>
<td>A2</td>
<td>56.18%</td>
<td>61.80%</td>
<td>69.67%</td>
<td>62.36%</td>
<td>83.71%</td>
<td>87.08%</td>
<td>87.64%</td>
</tr>
<tr>
<td>A3</td>
<td>51.46%</td>
<td>53.40%</td>
<td>69.90%</td>
<td>56.31%</td>
<td>83.50%</td>
<td>81.55%</td>
<td>84.47%</td>
</tr>
<tr>
<td>A4</td>
<td>56.67%</td>
<td>59.44%</td>
<td>60.56%</td>
<td>63.33%</td>
<td>79.44%</td>
<td>80.00%</td>
<td>83.89%</td>
</tr>
<tr>
<td>X</td>
<td>52.93%</td>
<td>56.37%</td>
<td>68.17%</td>
<td>58.79%</td>
<td>81.80%</td>
<td>82.30%</td>
<td>84.71%</td>
</tr>
</tbody>
</table>

The analysis given above indicates that a combination of two structured control strategies can account for approximately 80% of all activity transitions, but it does not discriminate between the two possible combinations of Breadth-First and Depth-First versus Children-First and Depth-First control strategies. Further analysis shows that the contribution to the Breadth-First score in both Tables 4.2 and 4.3 comes almost exclusively from the movement between siblings. This also happens to be the first item.
Global Control Strategies of Prolog Programmers

of the Children-First's definition. However, the predictions of the two control strategies deviate once all the children of a node have been recognised. A designer using a Breadth-First control strategy would immediately start describing any remaining complex siblings, while a designer using a Children-First control strategy would start describing the children of the node. Inspection of the keystroke protocols reveals that whenever a node that had just been described that has complex children and the designer was not employing a Depth-First control strategy, the designer always chose to describe one of these complex children rather than describing a complex sibling of the node. This indicates that designers used a combination of the Children-First and Depth-First approach rather than a combination of the Breadth-First and Depth-First approach.

Figure 4.1: Observed Depth at which A4 Worked across Transitions

Figure 4.2: Hypothetical Decomposition using a Breadth-First Control Strategy

Whether designers used a Depth-First strategy in combination with a Breadth-First or a Children-First control strategy can also be demonstrated graphically, as illustrated by
the graphs for A4 shown in Figure 4.1 (graphs for the activity transitions of the three other designers are shown in Appendix D).

The graph plots the transitions between nodes, and shows the level at which these nodes appear in the final goal hierarchy. If A4 used a Breadth-First control strategy, she would typically move from one node to another at the same level of the hierarchy and consequently remain at the same level in the hierarchy for prolonged periods (as illustrated in Figure 4.2). The absence of long horizontal lines in the graph shown in Figure 4.1 indicates that the designer rarely if ever used a Breadth-First control strategy at any stage. Instead, A4 worked at one level of the hierarchy for a number of transitions, but never for prolonged periods. This is a characteristic of the Children-First control strategy (as illustrated in Figure 4.3). The graph has a number of sharp peaks, indicating that the designer jumped up a level followed by an immediate descent.

![Figure 4.3: Hypothetical Decomposition using a Children-First Control Strategy](image)

4.5.2. **Structured-Divergent Activities**

Under the weaker definition of structure-conformance, 82% of the observed activity transitions were accounted for by a mixture of Children and Depth-First control strategies. The remaining 18%, structured-divergent, transitions were analysed further by examining the verbal protocols and the goal hierarchies to determine their nature and possible cause. They were found to consist of eight types, which are detailed below, and whose relative frequencies are given in Table 4.4.
## Table 4.4: Frequency and Absolutes for each Type of Structure-Divergent Transitions

<table>
<thead>
<tr>
<th>Type</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent not Coded</td>
<td>9.71%</td>
<td>1.69%</td>
<td>6.80%</td>
<td>3.89%</td>
<td>5.52%</td>
</tr>
<tr>
<td></td>
<td>(17)</td>
<td>(3)</td>
<td>(7)</td>
<td>(7)</td>
<td>(34)</td>
</tr>
<tr>
<td>Jump-Back</td>
<td>2.86%</td>
<td>0.56%</td>
<td>4.85%</td>
<td>3.89%</td>
<td>3.04%</td>
</tr>
<tr>
<td></td>
<td>(5)</td>
<td>(1)</td>
<td>(5)</td>
<td>(7)</td>
<td>(18)</td>
</tr>
<tr>
<td>Analogy</td>
<td>1.71%</td>
<td>1.69%</td>
<td>1.94%</td>
<td>2.22%</td>
<td>1.89%</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(3)</td>
<td>(2)</td>
<td>(4)</td>
<td>(12)</td>
</tr>
<tr>
<td>Debugging</td>
<td>1.71%</td>
<td>3.37%</td>
<td>0.00%</td>
<td>2.22%</td>
<td>1.83%</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(6)</td>
<td>(0)</td>
<td>(4)</td>
<td>(13)</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>1.71%</td>
<td>2.25%</td>
<td>0.97%</td>
<td>1.11%</td>
<td>1.51%</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td>(1)</td>
<td>(2)</td>
<td>(10)</td>
</tr>
<tr>
<td>Pre-requisite</td>
<td>0.00%</td>
<td>0.56%</td>
<td>0.97%</td>
<td>3.89%</td>
<td>1.36%</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(1)</td>
<td>(1)</td>
<td>(7)</td>
<td>(9)</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0.00%</td>
<td>0.56%</td>
<td>1.94%</td>
<td>2.78%</td>
<td>1.32%</td>
</tr>
<tr>
<td></td>
<td>(0)</td>
<td>(1)</td>
<td>(2)</td>
<td>(5)</td>
<td>(8)</td>
</tr>
<tr>
<td>Bottom-up</td>
<td>1.71%</td>
<td>2.25%</td>
<td>0.97%</td>
<td>0.00%</td>
<td>1.23%</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(4)</td>
<td>(1)</td>
<td>(0)</td>
<td>(8)</td>
</tr>
<tr>
<td>Total</td>
<td>19.43%</td>
<td>12.92%</td>
<td>18.45%</td>
<td>20.00%</td>
<td>17.70%</td>
</tr>
<tr>
<td></td>
<td>(34)</td>
<td>(23)</td>
<td>(19)</td>
<td>(36)</td>
<td>(112)</td>
</tr>
</tbody>
</table>

### 4.5.2.1. Transition to a node whose parent has not been coded, but which has been recognised

The most common type of diversion present for all four designers is the coding of a sub-goal without its parent being coded, even though the parent node has already been recognised. This type of transition occurs when the designer decides that the parent node does not need to be coded. An example of this occurred when A4 was designing a goal to display the results of the program. The goal consisted of four sub-goals to display the total number of cars, display the average number of cars per hour, display the busiest 60 minutes, and display the lightest 60 minutes. Below these sub-goals were simpler sub-goals. When A4 recognised these intermediate sub-goals, she did not code them. Instead A4 implemented all four sub-goals within the same routine.
4.5.2.2. Transition that jumps back to a structured approach after a divergence

The second most common type of structured-divergent transition is moving to a node in order to return to a structured control strategy. When designers have deviated from a structured approach, they often return to a node which would have conformed to the structured approach had the designers not diverted their attention. A typical example is when designers deviate from the structured control strategy to code a pre-requisite action for a variable that they had been considering. After implementing the pre-requisite action the designers return to their previous area of interest.

4.5.2.3. Transition to amend an erroneous node

Debugging transitions occurred after the designer compiled the program and discovered that its behaviour was different from what was expected. After diagnosing the cause of the difference, and locating the goal(s) which was faulty, the designer makes the necessary alterations to the goal(s). During this experiment the participants made trivial errors that were cured by the alternation of simple goals. The participants were not engaged in extensive periods of debugging, so it was not possible to determine whether they would use structured control strategies for complex errors.

4.5.2.4. Transition to capitalise on an analogy

On a few occasions designers noticed that two goals were similar and consequently alternated between the two goals while solving them. This alternation often occurred when the designer was producing either a goal to display the busiest 60 minutes of traffic or a goal to display the lightest 60 minutes of traffic and noticed their similarity. For example, the following verbalisation occurs in the protocol of A1 (square brackets indicate keystroke activity):

"So, let's call it ... write block time, max start at, [writing a node to display the starting time of the busiest 60 minutes] and write block time, min start at, [writing a node to display the starting time of the lightest 60 minutes], whoops, newline, comma, [writing a node to display the appropriate text with the start time of the busiest 60 minutes], write 'the lightest 60 minutes started at' ..." – A1.

When the designers noticed similarity between a number of goals they did not always enter into this alternating behaviour. Instead they often completed the first goal, before starting on the next similar goal. Since the similar goals were always siblings in this task, producing similar goals sequentially conformed both to a Children-First control strategy and to a Depth-First control strategy.
4.5.2.5. Transition to implement a pre-requisite for the current goal

The fifth type of structured-divergence is implementing an instruction that is a pre-requisite for the current goal being pursued. For example, if the current goal uses a variable to count, a pre-requisite would be to initialise the variable to some value, normally zero. The initialisation often occurs within the parent of the current goal. In order for the designers to implement the initialisation they have to abandon temporarily the structured control strategy, by jumping up a level to add to the description of the current node's parent.

4.5.2.6. Transition to select a goal based on its ease of implementation

During the design process designers sometimes selected a goal that they judged as being easy to describe, i.e. selecting a sub-goal that is not complex. This often occurred when the designer had just starting on the project. For example:

"Well the first task, the number of cars that pass the camera during the day that's easy ... Um, I think I'll probably do that first, 'cause that seems like a fairly simple task. So lets have a [goal] called count_cars ..." — A2.

4.5.2.7. Transition to implement a higher level goal

On occasions designers might complete a goal which they had started without recognising its parent, which then required them to later insert its parent. Within a Top-Down control strategy, a goal should only be considered after its parent has been recognised. For example:

"So I want the total number of cars ... which we know, that's easy then I don't need to do any computation, I can put that straight in. So lets try and build up the top levels [goals]." — A2

Bottom-up transitions of this nature often occurred after a structure-divergent transition to an easy goal, i.e. when the easy goal had been completed and its parent goal was then described.

4.5.2.8. Transition to implement a post-requisite for the current goal

Typically a post-requisite in the Signals problem involves passing a variable through a number of higher level goals. For example, the result from the goal to calculate the busiest 60 minutes of traffic needs to be passed through a number of goals at higher levels, until it has been passed back down to the goal of displaying the starting time of the busiest 60 minutes. To insure that the variable is passed up correctly the designer may abandon the current goal, and concentrate on a higher level goal. For example:
Global Control Strategies of Prolog Programmers

"I have got to make sure [in] all my places where I recurse [to] actually remember to keep the right number of arguments, that's a classic..." – A4.

Pre-requisites and post-requisites can occur for actions other than variable use. For example, to read in data from a file, there is the pre-requisite of opening the file, and the post-requisite of closing the file.

4.5.3. Factors in Switching Between Global Control Strategies

The sections of the goal hierarchies where it was possible to identify which control strategy the designers were using are examined, to determine the factors that underlie switches between a Children-First control strategy and a Depth-First control strategy. When a goal consists of simple sub-goals, (i.e. the sub-goals that can not themselves be decomposed) the behaviour of a designer using a Depth-First control strategy or a Children-First control strategy are identical. However, in the cases where the sub-goals were themselves decomposed it is generally possible to determine whether the designers were using a Depth-First control strategy or a Children-First control strategy. By analysing these cases, two factors affecting the choice of strategy were determined, these were:

1. whether the sub-goals were disjunctives (sub-goals are disjunctive when only one of them needs to be executed), and
2. the difficulty of designing the sub-goals. (A measurement of a sub-goal’s difficulty is provided by the number of generations below the sub-goal.)

There were 30 instances where it is possible to determine whether the designers were using a Depth-First control strategy or a Children-First control strategy, and where the sub-goals to be described were not disjunctives. A Children-First control strategy was used in 15 instances, and a Depth-First control strategy was used in the other 15 instances. For each of the 30 instances we measured the average number of generations for complex sub-goals. The average number of generation of a sub-goal constructed by the Children-First control strategy was 21.4 (standard deviation was 7.0), while the average for the Depth-first control strategy was 9.6 (standard deviation was 5.4). Thus, it appears that whenever the designers encountered disjunctives they employed a Depth-First control strategy. In the situations where the designers were confronted

---

20 If a goal consists of only one complex sub-goal, and that is the last one to be recognised, it is impossible to determine which strategy the designer used.

21 Note that it is not propose that designers necessarily estimate the difficulty of a sub-goal via this route.
with conjunctive sub-goals their choice of control strategy appears to be due to the difficulty of the sub-goals as inferred from the Depth of the sub-goal's solution. Designers were likely to use a Children-First control strategy when the sub-goals were difficult, but a Depth-First control strategy when the sub-goals were easy.

4.5.4. Local Control Strategies

The solutions produced by the designers for this design task rarely contained goals which had more than one non-simple sub-goal. Consequently the designers did not often have a list of sibling sub-goals to prioritise. As mentioned before, designers used a Depth-First control strategy when they encountered disjunctive sub-goals. When employing a Depth-First control strategy the sub-goal that is chosen to be described next is always the one that has just been recognised. Thus, there is no need to have a local control strategy while using a global Depth-First control strategy. Analysis across the protocols of the four designers reveals 13 occasions when there were two or more complex conjunctive sub-goals to prioritise. In five of these 13 cases the designers used a Children-First control strategy, and consequently they had to use a local control strategy. The only local control strategy observed in the verbalisations was one based on the order in which the sub-goals appeared in the program text. There was no evidence that designers assessed either the short-term advantage or the long-term advantage of each sub-goal in order to select the next one to complete.

4.6. Discussion

This study suggests that expert Prolog programmers predominately use a combination of Children-First and Depth-First strategies. If the assumption was made that a Breadth-First strategy is the only structured strategy a designer can use, then the conclusion would have been that 47% of transitions were structured-divergent. With such an apparent high percentage of structured-divergent transitions, it is not surprising that previous researchers concluded that designers were using an Opportunistic strategy. The percentage of structured-divergent transitions is reduced to 18% when the possibility of expert designers using alternative structured control strategies is recognised.

However there are still a large minority of transitions that do not conform to a structured control strategy. Whether these transitions were the result of Opportunistic behaviour depends upon the definition offered for Opportunism. Each structured-divergent transition has been examined to discover the purpose of that transition, which resulted in the identification of eight different types. It is difficult to argue that seven of these eight types are truly Opportunistic, in the sense of unexpected opportunities that present themselves to the designer. In the main, they reflect activities such as the description and completion of goals that had earlier been recognised, activities that...
Global Control Strategies of Prolog Programmers

returned to a structured approach, debugging, and the Depth-First implementation of a
goal. The exception is in transitions to capitalise upon analogies, where designers used
a partial solution for one problem to guide the production of a solution to similar
problem.

We have observed expert Prolog program designers using structured control
strategies, but is this result generalisable to experts with other programming languages?
As noted by Green, Bellamy, & Parker (1987) the nature of the task language and the
device language can contribute to the programmer’s performance in generating code. In
the observational study the task language was Prolog, and the device language was the
editor in LPA MacProlog. Prolog is a highly modularised language. In such a
language the only interactions that can occur between modules are through their
parameters (with the exception of ‘red cuts’ in Prolog, and the use of its database). This
means that the recognition of a module only affects the recognition of sibling
modules. The internal design of a module has no effect on the internal design of
another module. So in modularised languages, the danger of distant goals interacting
with each other is negligible. As a consequence within the modular programming
paradigm, the Breadth-First strategy has no advantage over the Children-First strategy.
For highly modularised languages a Children-First strategy is more advantageous than a
Breadth-First strategy.

4.6.1. Languages and Editors.
The opposite would be predicted for less modularised languages such as C. In C, the
scope of a variable is not necessarily limited to a module. This means that two program
components that are distant in the goal decomposition hierarchy can share the same
variable, so the internal calculations of one module will affect the calculations in the
other module. To combat unforeseen interactions it would be wise for the designers to
use the Breadth-First strategy rather than Children-First.

It is also probable that the device language (i.e. the editor) affect their choice of
strategy, since the ability to access information can support or inhibit a control strategy.
One property of an editor is its ‘access window’, that is, is the range of text that can be
readily accessed. Text that is easily accessible in one editor may be difficult to access in
another. For example, the editor of the task language ProGraph’s displays each
module in a separate window. Within a module’s window its sub-modules are
arranged. It is only possible to access a sub-module’s window from its parent’s
window. In such an editor Breadth-First is a difficult strategy to maintain, because it is
difficult to move between two goals of the same generation that are not closely related.
It is not a simple matter to access the next goal on the same level of the hierarchy, to do
so can involve going through a number of intermediate goals. Therefore, the editor
used for ProGraph would strongly constrain the programmer to using either a Children-First or a Depth-First strategy.

An editor's access window could be designed to support the use of a Breadth-First strategy. Such an editor would offer the facility to jump from the current goal that has just been described to any other goal on the same level of the hierarchy, no matter how distant the relationship between the two goals. However, most editors do not explicitly support any strategy. For example, the access window for the emacs text editor allows the programmer to view code only in sequential order. Since it is the programmers who choose the sequential order of the code, and programmers can order the code in a manner that is in line with the Breadth-First strategy or in a manner that is in line with the Children-First strategy, it follows that standard text editors equally support both strategies.

4.6.2. Design Issues

Although the nature of the task language and of the device language may affect the choice of the global control strategy, there are also other factors. From the observational study it has been noticed that the nature of the problem can affect the choice of global control strategy. Both the complexity of a goal and the relationship with its siblings can affect the choice of control strategy.

It seems reasonable to suppose that expert designers choose the most cost-effective strategy for producing a good design. Cost-effectiveness is determined by the ease of using a strategy against the likelihood of that strategy producing incompatible goals and the perceived cost of producing and rectifying incompatible goals:

$$\text{economic cost} = \frac{\text{ease of strategy}}{\text{chance of incompatible goals} \times \text{cost of producing and rectifying incompatible goals}}$$

The likelihood for a Children-First strategy and a Breadth-First strategy of producing incompatible goals is the same if the heuristic strategy of modularisation is used. A Children-First strategy is more likely to produce incompatible goals than a Breadth-First strategy if an alternative heuristic strategy is used.

If a goal consists of disjunctive sub-goals, the dependency of the sub-goals on each other is low. If designers encounters a goal that consists of disjunctive sub-goals, they are likely to employ a Depth-First strategy as was observed. When the sub-goals are disjunctive, all three structured strategies have an equal chance of producing incompatible sub-goals. Therefore advantage of a Children-First strategy over a Depth-First strategy is cancelled when faced with disjunctive sub-goals. A Depth-First strategy remains easier to implement than a Children-First strategy, thus making a Depth-First strategy preferable.

The cost of producing and rectifying incompatible goals is dependent on the difficulty of the goals. Within this study the difficulty of a goal was measured as the
Global Control Strategies of Prolog Programmers

Depth of sub-goals below the goal. If a goal consists of sub-goals which are not complex, the designer may choose a Depth-First strategy as the cost of producing and recovering from a mistake is low, even though the chance of a mistake may be high. With complex sub-goals the cost of producing and recovering from a mistake is high. The use of a Depth-First strategy for complex sub-goals is prohibitive as both the chance and the cost of a mistake are high.

In addition to observing the global control strategies used by designers I attempted to identify the local control strategies used to prioritise the order in which sibling sub-goals were pursued. Visser (1990) proposed that experts use a short-term (immediate) notion of cognitive cost when selecting between competing goals. Ball and Ormerod (1995) proposed that experts use a long-term (not necessarily immediate) notion of cost-effective design. No evidence that experts evaluated competing goals on either short-term or long-term considerations was found. Instead, their choice was dependent upon the order in which the goals were mentioned in the program text. This is also, in Prolog, the order in which goals are executed. In truth, it is possible to claim to have provided a sufficient test of competing explanations of local goal prioritisations, since the design problem and task language used in this study did not generate sufficient sibling sub-goals for such local prioritisations to be a significant issue. However, the approach adopted in this study offers a promising method for analysing design tasks with still larger goal hierarchies.
Chapter 5

The Global Control Strategies of Expert C Programmers
Contents of Chapter

This chapter reports a replication of the experiment described in the previous chapter, except that expert C programmers were observed instead of expert Prolog programmers. The aim of the experiment is to gauge whether the global control strategies used by Prolog programmers is a phenomenon caused by the Prolog language, or whether the behaviour of Prolog programmers is similar to programmers using a different programming language. The occurrence of Opportunistic behaviour was also observed and compared against the results of the Prolog study.

The results of the C study showed that C programmers behaviour similar to Prolog programmers. Both C and Prolog programmers used a combination of the Children-First and Depth-First control strategies, although the C programmers were more likely to diverge from the Top-Down structured approach. The extra divergent behaviour of the C programmers is restricted to extra debugging activities and extra adding pre-requisites for the current problem. The increase in both divergent activities is caused attributes of the C language.

When Opportunistic behaviour occurred was examined. The majority of Opportunistic behaviour occurred in clusters. Clusters of Opportunities occur at particular stages of programming, for example at the start of solving a problem, changing the solution to a problem, and debugging. The types of Opportunistic behaviour that occur within clusters are different from the Opportunistic behaviours that occur in isolation. The type of Opportunistic behaviour that typically follow after particular types of Opportunistic behaviour is also examined, for example the next Opportunistic transition to occur after a debugging transition is often another debugging transition.
5. The Global Control Strategies of Expert C Programmers

5.1. Introduction

The results of the study on experienced Prolog programmers indicate that the programmers generally use either a Children-First strategy or a Depth-First strategy depending on the complexity of the problem to be solved. These two strategies accounted for over 82% of all transitions from one part of the problem to another.

The previous study concentrated on one particular programming language, Prolog. It is reasonable to ask whether the results collected in that study are an artefact of the language, or whether the results can be generalised to other languages. Prolog is an idiosyncratic programming language. Prolog does not have variables, it has a very restrictive syntax, with very few keywords, and its foundation is in first-order predicate logic. As a result it is questionable whether the results from the Prolog study can be generalised to any other programming languages.

The study described in this chapter was undertaken to investigate how generalisable the results from the Prolog study are. If the performance of programmers using two dissimilar languages can be predicted by the same model, then the argument that the model’s scope extends across languages in general is made stronger. The greater the difference between languages whose use of control strategies has the same explanatory basis in the proposed theoretical model, then the greater the scope and strength of the model.

The C language (Schildt, 1990) has many differences from the Prolog language (Bratko, 1990), and as such it is ideal for testing the scope and strength of the proposed model. Where Prolog has facts (similar to constants, whose value cannot be varied), C has variables. Prolog has a sound theoretical foundation built on logic, whereas C evolved as a procedural language from BCPL via B, and its rationale is to overcome shortcomings of its predecessors. C is a middle level language, i.e. it is not strongly typed. Whereas Prolog is a high level language. In addition, Prolog has extremely few keywords, while C has 32 keywords, plus a large library of standard functions. Finally, modules in Prolog are closed whilst C’s modules are open. Given the many differences between C and Prolog, C is an ideal language to test how generalisable the

---

22 In most languages stored data has a value and a data type (e.g. integer, string, etc.). In a weak type language such as C, stored data are not strongly linked to a particular data type. For example an integer in C can be used as a character or a number.

23 A closed module is self contained and can only communicate with other sections of the program via parameters.
results of the Prolog study are. The differences between the two languages are summarised in Table 5.1.

<table>
<thead>
<tr>
<th>Features</th>
<th>Prolog</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paradigm</td>
<td>Logic/Declarative</td>
<td>Procedural</td>
</tr>
<tr>
<td>Level</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Variables</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modules</td>
<td>Closed</td>
<td>Open</td>
</tr>
<tr>
<td>Keywords</td>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

5.1.1. Hypothesis

To produce a hypothetical account of C programmers' behaviour it is necessary to provide a summary of the explanation offered for Prolog programmers' behaviour. From the results of the study reported in the previous chapter, it appears that Prolog programmers largely produce their solutions Top-Down. As argued in Chapter 2 there are three strategies for producing a solution Top-Down (Breadth-First, Children-First, and Depth-First). Each of the strategies has its own advantages and disadvantages. There are circumstances for each strategy where it is the most advantageous strategy, and other circumstances where it is the most disadvantageous one.

The factors that influence the use of the strategies are (1) the programmer's knowledge, (2) the complexity of a problem, and (3) the openness of the language. Given that Prolog is a closed language, only the Depth-First and the Children-First strategies are truly advantageous. This is supported by the data from the previous study.

In the model developed in the previous chapter, the programming language can affect the strategy chosen by its degree of openness. The more open a language is then the more likely the internal processes of a module is to be influenced by the internal processes of other modules. C is an open language. Therefore it is necessary for C programmers to identify the sources of possible interference. Neither Children-First nor Depth-First strategies enable the programmer to identify exhaustively potential interferences, unlike a Breadth-First strategy. To produce a well-defined solution, C programmers should ideally use a Breadth-First control strategy.

24 C and Prolog I/O keywords were not counted to simply comparison.
Therefore a study was carried out to investigate the strategies used by expert C programmers, replicating the approach taken in the previous study. The hypothesis was that experienced C programmers will use the Breadth-First global control strategy, except for simple sub-problems, where they will employ the Depth-First global control strategy.

5.2. Method

5.2.1. Participants

Five experienced C programmers participated in this study. The first programmer (B1) was a computer technician. He had 15 years experience of designing and implementing programs in a variety of languages. He had used C for 14 years. The second participant (B2) was a Computer Science PhD student who had ten years experience of designing and implementing programs, and had used C for five years. The third participant (B3) was a lecturer in Computer Science. He had 12 years programming experience, during which time he had used C for four years. The fourth programmer (B4) was also a PhD student in Computer Science. B4 had been designing and implementing programmers for eight years. In seven of those years he had been using C. The final programmer (B5) was the least experienced programmer. He was a PhD student in Computer Science, with two years experience of C, and three years of programming in general.

5.2.2. Task and Materials

Each designer was presented with standard introductions (Appendix E.1.) detailing what they would be asked to do. When they had read the standard introductions, they were given the specification of a practice task (Appendix E.2.) to familiarise them with concurrent verbalisation. Once the practice task was completed, the participants were presented with the specification to the main task (Appendix E.3.). The main task involved handling a series of signals about traffic flow at a number of road junctions. The task included calculating and displaying statistics about traffic use at each road junction.

The participants were required to edit the solution in the editor TextRec on an Apple Macintosh. The participants were allowed to use Symantec's Think C to test their solutions. The programmers also had access to pen, paper, and a C manual.

5.2.3. Procedure

The standard instruction asked the participants to verbalise everything they were thinking during the experiment. Furthermore the experimenter prompted them if they became silent for more than 30 seconds.
The participants were then given a short programming task to familiarise themselves with thinking-aloud, and with TextRec. After completing the practice task, the participants were given the main task to complete. They were informed that they could use paper to jot down ideas and that the experimenter would answer any questions regarding the input/output of data from a file. The participants also had access to a C manual.

During the main task, verbal reports were recorded along with keystroke actions. Any piece of paper used by the participants was also collected as data. The participants were given two hours to complete both the practice task and the main task.

### 5.3. Results and Discussion

#### 5.3.1. Global Control Strategies

The functional definition of conforming to a global control strategy used in the previous Prolog study was reapplied to the performance of the C programmers. This enabled meaningful comparisons between Prolog and C programmers.

The final definition of conforming used in the Prolog study was to compare the liberal and conservative definitions and to resolve any difference by using qualitative data. When this method of measuring conformity was applied to the behaviour of C programmers the following results shown in Table 5.2 were obtained.

<table>
<thead>
<tr>
<th></th>
<th>Breadth &amp; Depth</th>
<th>Children &amp; Depth</th>
<th>Children &amp; Breadth</th>
<th>Breadth &amp; Children</th>
<th>All Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>47.24%</td>
<td>60.24%</td>
<td>69.29%</td>
<td>61.02%</td>
<td>81.50%</td>
</tr>
<tr>
<td>B2</td>
<td>36.92%</td>
<td>48.13%</td>
<td>64.49%</td>
<td>49.53%</td>
<td>74.30%</td>
</tr>
<tr>
<td>B3</td>
<td>55.45%</td>
<td>57.92%</td>
<td>66.83%</td>
<td>61.88%</td>
<td>78.22%</td>
</tr>
<tr>
<td>B4</td>
<td>51.11%</td>
<td>59.26%</td>
<td>68.15%</td>
<td>61.48%</td>
<td>78.89%</td>
</tr>
<tr>
<td>B5</td>
<td>48.45%</td>
<td>57.73%</td>
<td>57.73%</td>
<td>59.28%</td>
<td>71.65%</td>
</tr>
<tr>
<td>X</td>
<td>47.88%</td>
<td>56.88%</td>
<td>65.70%</td>
<td>58.82%</td>
<td>77.25%</td>
</tr>
</tbody>
</table>

The main hypothesis of this study was that C programmers would use a Breadth-First control strategy with possibly some Depth-First behaviour. The reason for this hypothesis was that C is an open language, whereas Prolog is a closed language.

As can be seen from the average conformity of each control strategy, C programmers follow a mixture of Children-First and Depth-First control strategies. In other words, C programmers use the same control strategies as the one used by Prolog.
Global Control Strategies of C Programmers

programmers. Thus, the hypothesis was not confirmed, instead it appears that the use of Children-First and Depth-First control strategies is observed across languages.

It is possible that the C programmers attempt to resolve the problems associated with open languages by employing techniques used with closed languages. From examination of the final solutions produced by the five designers it is clear that the functions they produced only communicate among each other by passing data as parameters. As a consequence the designers could concentrate on each function without having to worry about interference from other functions.

When designers produce closed solutions, the Children-First control strategy is as good as the Breadth-First control strategy for making sure that the final solution produced fulfils the specified requirements, while the Breadth-First control strategy requires more cognitive resources than the Children-First control strategy. Therefore the Children-First control strategy has an advantage over the Breadth-First control strategy, which will lead programmers to select the Children-First control strategy in favour of the Breadth-First control strategy.

5.3.1.1. Comparison between the Global Control Strategies of C and Prolog Programmers

Since Prolog and C programmers employ similar control strategies, it is legitimate to compare the consistency with which Prolog and C programmers conform to a mixture of Children-First and Depth-First control strategies.

Table 5.3: Percentage of Transitions Conforming to Children-First and Depth-First Control Strategies

<table>
<thead>
<tr>
<th>Participants</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>80.57%</td>
</tr>
<tr>
<td>A2</td>
<td>87.08%</td>
</tr>
<tr>
<td>Prolog A3</td>
<td>81.55%</td>
</tr>
<tr>
<td>A4</td>
<td>80.00%</td>
</tr>
<tr>
<td>X</td>
<td>82.30%</td>
</tr>
<tr>
<td>B1</td>
<td>81.50%</td>
</tr>
<tr>
<td>B2</td>
<td>74.30%</td>
</tr>
<tr>
<td>C B3</td>
<td>78.22%</td>
</tr>
<tr>
<td>B4</td>
<td>78.89%</td>
</tr>
<tr>
<td>B5</td>
<td>71.65%</td>
</tr>
<tr>
<td>X</td>
<td>77.25%</td>
</tr>
</tbody>
</table>
Global Control Strategies of C Programmers

Table 5.3 shows a comparison of mean conformance to a mixture of Children-First and Depth-First global control strategies. Although remarkably similar in degree of adherence to this approach, C programmers seem to conform less often to the two control strategies than do the Prolog programmers. The average difference is approximately 5%.

5.3.2. Divergence from the Global Control Strategies

Each individual transition that diverged from the Children-First and Depth-First control strategies was examined to determine the nature of that transition. The classification of Opportunism that was determined in the Prolog study was applied to the divergent transitions of C programmers.

There were four Opportunistic Transitions of the designer B1 that could not with any accuracy be assigned to one of the eight categories, and there was not sufficient information from the protocols to identify an alternative reason for the divergent behaviour.

Table 5.4: Occurrence of Various Opportunistic Transitions

<table>
<thead>
<tr>
<th>Opportunities</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Not Coded</td>
<td>2.76%</td>
<td>3.27%</td>
<td>4.95%</td>
<td>5.19%</td>
<td>7.73%</td>
<td>4.67%</td>
</tr>
<tr>
<td>Jump Back</td>
<td>2.36%</td>
<td>5.14%</td>
<td>4.95%</td>
<td>3.33%</td>
<td>4.64%</td>
<td>3.97%</td>
</tr>
<tr>
<td>Analogy</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>1.11%</td>
<td>2.58%</td>
<td>0.71%</td>
</tr>
<tr>
<td>Debugging</td>
<td>2.36%</td>
<td>6.07%</td>
<td>3.47%</td>
<td>4.44%</td>
<td>1.55%</td>
<td>3.61%</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>3.15%</td>
<td>4.21%</td>
<td>0.50%</td>
<td>0.74%</td>
<td>3.09%</td>
<td>2.29%</td>
</tr>
<tr>
<td>Prerequisite</td>
<td>3.54%</td>
<td>6.07%</td>
<td>6.93%</td>
<td>4.44%</td>
<td>6.19%</td>
<td>5.29%</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.37%</td>
<td>1.03%</td>
<td>0.26%</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>2.76%</td>
<td>0.93%</td>
<td>0.99%</td>
<td>1.48%</td>
<td>1.55%</td>
<td>1.59%</td>
</tr>
<tr>
<td>Not Known</td>
<td>1.57%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.35%</td>
</tr>
<tr>
<td>Total</td>
<td>18.50%</td>
<td>25.70%</td>
<td>21.78%</td>
<td>21.11%</td>
<td>28.35%</td>
<td>22.75%</td>
</tr>
</tbody>
</table>

5.3.2.1. Comparison of Opportunities between C and Prolog Programmers

Whilst structured design still predominated, the occurrence of Opportunistic Transitions was greater for C programs than Prolog programs. The occurrence of each type of Opportunism, as illustrated in Table 5.5, seems to be similar between C and Prolog programming, with two exceptions. These exceptions are Debugging Opportunities.
and Prerequisite Opportunities, both of which occur with higher frequency during C programming than during Prolog programming.

When one considers the difference between the C and the Prolog language, it is not surprising that the occurrence of Debugging and Prerequisite Transitions are greater in C than it is in Prolog. Two major differences between C and Prolog are: (1) C has variables, while Prolog only has facts (similar to constants); and (2) C's syntax is more complex than Prolog's.

A major cause of Prerequisite Transitions is the handling of data. To handle data within C involves the use of variables that need to be declared and initialised. If a C programmer introduces a new variable without having provided the necessary support, the programmer then has to engage in Prerequisite Transitions. As Prolog does not have variables, there is no necessity to declare and initialise them, and therefore there is less reason for a Prolog programmer to make Prerequisite Transitions than there is for a C programmer.

<table>
<thead>
<tr>
<th>Opportunities</th>
<th>C</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Not Coded</td>
<td>4.67%</td>
<td>5.52%</td>
</tr>
<tr>
<td>Jump Back</td>
<td>3.97%</td>
<td>3.04%</td>
</tr>
<tr>
<td>Analogy</td>
<td>0.71%</td>
<td>1.89%</td>
</tr>
<tr>
<td>Debugging</td>
<td>3.62%</td>
<td>1.83%</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>2.29%</td>
<td>1.51%</td>
</tr>
<tr>
<td>Prerequisite</td>
<td>5.29%</td>
<td>1.36%</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0.26%</td>
<td>1.32%</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>1.59%</td>
<td>1.23%</td>
</tr>
<tr>
<td>Not Known</td>
<td>0.35%</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>22.75%</td>
<td>17.70%</td>
</tr>
</tbody>
</table>

The complexity of syntax may cause the difference in the amount of debugging required. As a programming language's syntax increases in complexity, then the possibility of errors occurring increase. As C is more syntactically complex than Prolog, the C programmers are consequentially more likely to produce errors than are the Prolog programmers. This will entail C programmers having to amend more errors (i.e. engage in more debugging) than Prolog programmers.
5.4. When Do Opportunities Occur?

5.4.1. Prediction

With the exception of Debugging Opportunism, Opportunism is generally considered to be the recognition of a problem that the designer knows the solution to, and where the designer immediately proceeds to produce that solution. With such a definition it is clear that the occurrence of Opportunism depends on two factors:

- the number of solutions the designer knows, i.e. the expertise of the designer, and
- the number of goals left to be solved, i.e. the completeness of the problem.

The first factor, the expertise of the designer, increases with experience. As the designer produces an artefact, he may produce a solution that was not initially part of his/her repertoire, but which he/she may now in the future use Opportunistically. However as the designer becomes more experienced, then the less likely is the designer to have to solve a problem that requires a novel solution.

Provided the designer is an expert, this factor will remain largely constant during the design of an artefact, with a slight possibility that this factor may increase. The second factor, the number of goals to be solved will change dramatically during the course of the design stage.

![Progression of Constructing Solution](image)

**Figure 5.1:** Possibility for Opportunism at Stages of Design of Solution

If it is assumed that the first factor is principally constant, with the second factor causing most of the variability in the scope of Opportunism during a programming task then the following pattern of occurrence of Opportunism would be predicted:
Global Control Strategies of C Programmers

- during the first stages of design the scope of Opportunistic behaviour will increase as the number of recognised goals increase;
- gradually the number of goals being recognised will slow down, to the point were the number of goals being recognised is smaller than the number of goals being described. So after the initial explosion in the scope for Opportunism, the increase in Opportunism will slow down and eventually decrease; and
- towards the end of the design stage, the number of goals left to be described is very small, and therefore the scope of Opportunism is also very small.

Figure 5.1 provides a graphical representation of Opportunism Transitions occurring principally during the middle stage of program development, with conforming transition occurring with greater frequency at the beginning and end of program development.

5.4.2. Observation

With this prediction is mind, the percentage of divergent transitions was examined. For each designer the percentage of divergence was calculated for each twentieth percentile. On average each percentile accounts for 11.34 transitions.

A scatter graph for each designer indicates that there is no function determining occurrence of Opportunism against time. For designer B1 there was a linear relationship between time and Opportunism, but this did not occur for any of the other designers. The scatter graph's of B2, B4 and B5 are shown in Appendix F. Note that B1 showed the least Opportunistic behaviour out of all the C programmers.

![Figure 5.2: Opportunism at Stages of B1's Design of Solution](image-url)
The picture that has emerged is different to one that would be predicted from opportunism being a function of analogy. An alternative explanation is required to explain the observed results.

5.4.3. A Secondary Analysis

The main problem with examining transitions by grouping them into percentiles is that it provides a very grainy view of the data. Changes that may happen in the occurrence of opportunism, do not necessarily happen at the boundary of a percentile where any change is most observable. In addition, the period of change does not necessarily fit within the range of the percentiles.

An alternative approach of looking at opportunism occurring in clusters was employed, and to examine the various stages of the design process at which clusters of opportunism occurred. The first requirement is a definition for a cluster of opportunistic transitions. The following definition was constructed:

- a cluster must start with an opportunistic transition,
- a cluster must finish with an opportunistic transition,
- more than 50% of the transitions within a cluster must be opportunistic,
- a cluster must consist of four or more transitions, and
- a cluster of opportunism must not contain a cluster of transitions conforming to a global control strategy.

The definition of a cluster of transitions conforming to a global control strategy was based on the definition of a cluster of opportunism, i.e:
Global Control Strategies of C Programmers

• it must start with a Conforming Transition,
• it must finish with a Conforming Transition,
• more than 50% of the transitions within it must be conforming,
• a cluster must consist of four or more transitions, and
• a cluster of transitions conforming to a global control strategy must not contain a cluster of Opportunism.

The definition of an Opportunistic cluster was developed so that it contained the greatest possibly number of Opportunistic Transitions, while reducing the possible number of Conforming Transitions. It was decided that a cluster required to include four or more transitions so that when just two Opportunistic Transitions happened to occur in close proximity they were not considered to be a cluster. It was felt that two Opportunistic Transitions occurring in proximity to each other could be due to coincidence, but that three or more Opportunistic Transitions occurring in proximity was unlikely to be coincidental.

Figure 5.4: Clusters of Opportunism

Between the five designers, a total of 28 Opportunistic Clusters were identified when using the above definitions. Within the 28 clusters, 58.80% of Opportunistic Transitions occur, while 90.83% of Conforming Transitions occur outside an Opportunistic Cluster. The percentage of transitions within an Opportunistic Cluster that are Opportunistic is 65.82%. While, the percentage of transitions outside an Opportunistic Cluster that are Opportunistic is 11.75%.

Both the Verbal Protocols and the Keystroke Protocols that took place during a cluster of Opportunism were examined. From this examination it was discovered that a cluster coincides with one or more of six activity types. The six activity types are:

• Beginning to solve the problem,
• Beginning to solve a sub-problem,
• Changing the solution to a sub-problem,
• Analogy, reusing solution of a previous sub-problem,
Global Control Strategies of C Programmers

- Debugging the solution to a sub-problem, and
- No overall pattern to the Opportunistic Transitions.

Often the two activity types of changing a solution, and debugging a solution, also included an Opportunistic Transition to a new sub-problem once the designer has finished correcting, or debugging the current sub-problem.

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Problem</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Starting Sub-Problem</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Changing Solution</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Debugging Solution</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Analogy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No Overall Activity</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>32</td>
</tr>
</tbody>
</table>

Within the 28 Clusters, 32 activities were discovered. Each activity can be described by one of the six activity types listed above. The occurrence of activities among Clusters is shown in Table 5.6.

5.4.3.1. Beginning to solve the problem

The behaviour of four of the five designers initially diverged from a structured approach. During this stage the designers were uncertain of the solution they were going to produce for the problem presented to them. So during the initially stages the designers were considering how the problem could be divided by identifying the main tasks the solution needed to perform. The designer Opportunistically produced a skeleton of the solution, on which the details could later on be fleshed out. Below is the verbal protocols of B5 at the start of his design session. The numbers in square brackets correspond to numbers in the hierarchy diagram of the solution, as shown in Figure 5.5. The numbers also indicate the order in which the nodes of the hierarchy are considered.

"Okay, I'll just execute that [1]. ... Then [mumble] the way I do something like this would be to reload [2] or manipulate [3] and then [mumble]. And the save the program because of [mumble] ... Okay, um. So I'm always worried that negative[?] it could just be, this is a first processing called read file [4]. So I've got my file. And then we rate of traffic flow [5]. Which can be done in a single parse of the file,
without actually saving every single thing. The other part, sort [6]. There’s the surveys to order. The last one brings this up [7]. Then we output the surveys [8]. Right, I’m going to just write down what I think that will be. So at the start of each survey a detector transmit a record of its location in the form of the junction number where the traffic comes from. So, we’ve got. Followed by the junction number it goes to. So, we have two. And would these be stored, um, in what form, it says a binary file? [Yes] Is it on separate lines? [No] Whenever a vehicle, transmits number one. And at one second intervals, it transmits a number two. Transmits a zero at the end of survey. And another zero. [prompt] I just thinking, what things I’ve got to calculate. Just grasping the, sort of, detective high number. So the rate of traffic flow. And the length of the longest waiting period. [mumble] For each detector, we need to have: rate of traffic flow. Length of longest wait. The easiest data structured is the array [9]. And it probably wouldn’t need to be much storage space, because of one-way roads, and there would not be so much [mumble] So if we complete the number of possible, maximum number of detectors there are, could be [10]. Which is twenty-four. Twenty-four is the maximum number. If we keep a running count, while we’re reading in the, the actual survey, surveys, we get the number of surveys [11]. Now, I’m just going to read thorough the specifications to check I haven’t missed out. [prompt] I’m thinking about how I, if it’s possible for me do it. There is only one minor problem that I can see. And that is, I’m not quiet sure how to get just one single digit number. So what we want to do is go through, open the file [12]. And we are going to read in each of the surveys in turn [13]. And then within each survey there’s going to be different, we’re going to, um, count up the number of seconds [14]. And the number of cars, which will allow us to calculate the rate of traffic flow [15]. We need the longest wait. And we calculate that [16]. Each time, we receive the signal. [mumble] And then close the file [17].” – BS.

When the verbal protocol is viewed in conjunction with the hierarchy diagram it is clear that the designer is not focusing on one particular aspect of the problem. The designer is recognising the goals that the solution needs to be achieved, and gradually the relationships between these goals are established. Once the designers are satisfied with the overall outline, they will start to expand on it, in a more methodical Top-Down manner.
There was one instance of a Cluster of Opportunistic activity where designer B5 started to solve a sub-problem Opportunistically. This occurred shortly after he had been Opportunistically designing the general framework of the solution. The last
Opportunistic Transition to occur while the designer B5 was producing the skeleton of the solution was to the goal of sorting the data.

Designer B5 started to produce the solution to the problem of sorting data using a global control strategy. However designer B5 was unable to maintain a global control strategy because of his uncertainty. The designer then made a number of Opportunistic Transitions to identify essential nodes of the solution, around which the rest of the solution, could be produced.

5.4.3.3. Changing the solution to a sub-problem

The most common activity to occur within a Cluster was to change the solution that the designer had already produced.

There were two types of alteration that the designers made, these were either to the control flow, or to the data flow. Of the 13 instances where designers changed their solution, five were changes to the control flow, and eight were changes to the data flow.

This imbalance in favour of the predominance of data flow changes is caused by the method in which designers realised that they had to make changes to their solutions. The designers noticed that an alteration in the solution was required either (1) by reviewing the solution they had produced, and discovering a flaw in the solution; or (2) by producing the solution on the hoof, and realising the sub-solution they had just produced requires a number of prerequisites, which they had not included in their solution.

The first method of discovering a flaw in the solution is impartial to the discovery of a control flow error, or of a data flow error whereas the second method is more likely to result in the discover of a data flow problem, than a control flow problem.

Data flow problems are more likely to be uncovered on the hoof than control flow problem because the use of data often requires prerequisite to be performed, such as declaration and initialisation whereas a control flow structure can be introduced without any prerequisites (except for the data that the control structure uses, and this is a data flow problem).

Five of the alterations were instigated after the designers reviewed their solutions. Three of the five alterations were made to the control flow, while the remaining two alterations were made to the data flow.

The other eight alterations were performed as the designer was constructing a solution. Three of the eight changes were made to the control flow, with the other five changes being made to the data flow.

Figure 5.6 shows an example of designer B1 amending his solution once he realised that the solution only calculated the number of cars during a survey, but not the length of survey, nor the length of the longest period without any traffic. As a result
designer B1 needs to add a couple of tests, and a couple of calculations to his current solution.

Figure 5.6: Designer B1 Amending his Solution to the Design Problem

The solution B1 had initially produced was:

```
get number
if (number is 1) then
    increment survey length
```

After B1 had checked the solution, he amended it to:
5.4.3.4. Analogy, reusing solution to a previous sub-problem

Of the 32 separate Opportunistic activities, only one was an instance of a designer applying the solution from one sub-problem to that of another sub-problem. The instance of analogy was designer B5 using his solution of data passing from one sub-problem and applying the solution to other sub-problems.

```c
#include <stdio.h>

void input(int *rate, int *wait)
{
}

main()
{
    int rate[24], wait[24];

    input;
    sort;
    output;
}
```

After designer B5 had solved the problem of what data to pass to the sub-problem input, he then applied the solution to the sub-problems sort and output.
#include <stdio.h>

void input(int *rate, int *wait)
{

}

void sort(int *rate, int *wait)
{

}

void output(int *rate, int *wait)
{

}

main()
{
    int rate[24], wait[24];

    input;
    sort;
    output;
}

5.4.3.5. Debugging the solution to a sub-problem

The second most common activity to involve a Cluster of Opportunistic Transitions is the debugging of a solution. In this instance the activity of debugging refers to the designer making use of software tools that identify errors in the current solution. Often these errors were syntactical (e.g. typing mistakes), but they can also be semantic errors.

Debugging is instigated by a designer when he thinks that he has completed a solution to a sub-problem, and he wishes to test that the solution performs in a specified manner. By their nature, software tools first identify any syntactic errors. Only if there are no syntactic errors is the designer then able to use software tools to identify semantic errors. The designer therefore amends syntactic errors before proceeding on to the correction of semantic errors.
Global Control Strategies of C Programmers

When the designer believes he has corrected the error that the software tools have identified, he usually resubmits the solution to the software tools in order to see whether there are any more errors in the solution.

A debugging session is characterised by the designer submitting a solution to a software tool. If there is an error, he will try to correct the problem, and then resubmits the solution. This continues until the designer believes the solution to be free of errors, or until the designer hits an error that he can not currently correct.

5.4.3.6. No overall pattern to the Opportunistic Transitions

Out of the 32 instances of a Cluster of Opportunistic Transitions, over a fifth of them could not be associated with an overall activity. The six Opportunistic Clusters with no apparent overall activity seem to instances where the designers are following a global control strategy but made a number of Opportunistic Transitions in close proximity to each other, without have relationships to each other. However, these Opportunistic Transitions appeared as a Cluster as a result of the Cluster definition.

Table 5.7: Occurrence of Opportunistic Transitions in Clusters and in Isolation

<table>
<thead>
<tr>
<th>Type of opportunistic transitions</th>
<th>% of transitions that are opportunistic transitions</th>
<th>% of transitions that are opportunistic transitions and occur in isolation</th>
<th>% of opportunistic transitions that occur in isolation</th>
<th>% of transitions that are opportunistic transitions and occur in clusters</th>
<th>% of opportunistic transitions that occur in clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent not Coded</td>
<td>4.67%</td>
<td>1.94%</td>
<td>41.51%</td>
<td>2.73%</td>
<td>58.49%</td>
</tr>
<tr>
<td>Jump-Back</td>
<td>3.97%</td>
<td>2.03%</td>
<td>51.11%</td>
<td>1.94%</td>
<td>48.89%</td>
</tr>
<tr>
<td>Analogy</td>
<td>0.71%</td>
<td>0.09%</td>
<td>12.50%</td>
<td>0.62%</td>
<td>87.50%</td>
</tr>
<tr>
<td>Debugging</td>
<td>3.62%</td>
<td>0.97%</td>
<td>26.83%</td>
<td>3.65%</td>
<td>73.17%</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>2.29%</td>
<td>0.79%</td>
<td>34.62%</td>
<td>1.50%</td>
<td>65.38%</td>
</tr>
<tr>
<td>Pre-requisite</td>
<td>5.29%</td>
<td>2.91%</td>
<td>55.00%</td>
<td>2.38%</td>
<td>45.00%</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0.26%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.26%</td>
<td>100.00%</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>1.59%</td>
<td>0.53%</td>
<td>33.33%</td>
<td>1.06%</td>
<td>66.67%</td>
</tr>
<tr>
<td>Not Known</td>
<td>0.35%</td>
<td>0.09%</td>
<td>25.00%</td>
<td>0.26%</td>
<td>75.00%</td>
</tr>
<tr>
<td>Overall</td>
<td>22.75%</td>
<td>9.35%</td>
<td>41.09%</td>
<td>13.40%</td>
<td>58.91%</td>
</tr>
</tbody>
</table>

The number of transitions in a Cluster that has no overall pattern is between four and five, with a mean of 4.66. The average size of the other Clusters that could be associate with an activity is 8.23.
5.4.4. **Isolated Opportunistic Transitions**

Although the majority of Opportunistic behaviour occurred within a Cluster of Opportunism, 41.09% of Opportunistic behaviour occurred in relative isolation from the other Opportunistic Transitions. Do these isolate Opportunities differ from the Opportunities that occur in Clusters?

From Table 5.8, it appears that the distribution of Opportunistic Transition types is different when the transitions occur in isolation from when the transitions occur in Clusters.

<table>
<thead>
<tr>
<th>Type of opportunistic transitions</th>
<th>Number of opportunistic transitions that occur in isolation</th>
<th>Number of opportunistic transitions that occur in clusters</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Not Coded</td>
<td>22 (20.95%)</td>
<td>35 (23.03%)</td>
<td>57</td>
</tr>
<tr>
<td>Jump-Back</td>
<td>23 (21.90%)</td>
<td>22 (14.47%)</td>
<td>45</td>
</tr>
<tr>
<td>Analogy</td>
<td>1 (0.95%)</td>
<td>7 (4.61%)</td>
<td>8</td>
</tr>
<tr>
<td>Debugging</td>
<td>11 (10.48%)</td>
<td>30 (19.74%)</td>
<td>41</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>9 (8.57%)</td>
<td>17 (11.18%)</td>
<td>26</td>
</tr>
<tr>
<td>Pre-requisite</td>
<td>33 (31.43%)</td>
<td>26 (17.11%)</td>
<td>59</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0 (0.00%)</td>
<td>3 (1.97%)</td>
<td>3</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>6 (5.71%)</td>
<td>12 (7.89%)</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>105</td>
<td>152</td>
<td>257</td>
</tr>
</tbody>
</table>
Opportunistic Transitions to prerequisite nodes, and jumping back to previous nodes seem more likely to occur in isolation than in Clusters. The remaining transition types, except for 'Parent not Coded', seem more likely to occur in a Cluster than in isolation. 'Parent not Coded' seems as likely to occur in Clusters as it does in isolation.

5.4.4.1. Debugging Transitions

The reason for the difference in the distribution of the various types of Opportunistic Transitions between isolated Opportunities and Cluster Opportunities is that some types of Opportunistic Transitions are predisposed to occur either in a cluster or in isolation.

For example the Opportunistic Transition type, Debugging, rarely occurs in isolation, as (1) an error in the solution is often repeated elsewhere in the solution; (2) correcting an error may require more than simply altering one part of the solution; and (3) designers do not test their solution until a significant section has been produced, which may result is an accumulation of several separate errors that require debugging.

The five designers made a total of 261 Opportunistic Transitions, thus 256 Opportunistic Transitions are followed at some point by another Opportunistic Transition. In 251 of these pairs, the type of Opportunism of both partners is known. Of the known pairs, 28 pairs were instances of a Debugging Transition being followed by another Debugging Transition.

5.4.4.2. Prerequisite Transitions

Opportunistic Prerequisite Transitions seem to occur more often in isolation than in a Cluster. A Prerequisite Transition occurs when the designers realise that a section of the solution they have produced requires support that has not been produced in a previous section of the solution. Many of the Clusters are not concerned with uncovering the support required by parts of the solution. This limits the Opportunity of the Prerequisite Transitions to occur within Clusters.

Although the Prerequisite Transitions are the most common type of transitions they only occur in 15 of the 32 activities association with Clusters. Of these 15 activities, ten are making changes to the current solution, two have no overall pattern, and the remaining three concern the beginning of solving a problem or sub-problem. In addition to the limit of Prerequisite Transitions occurring within a Cluster, there are instances where Prerequisite Transitions can occur in isolation.

A Prerequisite Transition often occurs after the designers introduce a new variable to their solution. Variables need to be declared, and frequently initialised. When designers makes use of a variable for the first time, they have rarely included the required declaration and initialised in their solution. Therefore the designer has to make an Opportunistic divergence to make the necessary alternations, before jumping back to where the divergence was initiated. As the alternations are minimal they do not appear as a Cluster, but as being isolated.
5.4.4.3. Other Transitions

Of the remaining three types of Opportunistic Transitions that occur more often in a cluster than in isolation (Analogy, Post-requisite, Bottom Up), their occurrence is too rare for any interpretation to be made.

5.4.5. Predicting the Next Opportunistic Transition

The previous sections eluded to the possibility that certain types of Opportunistic Transitions are more likely to follow certain Opportunistic Transitions types than they are follow other types.

Table 5.9: Occurrence of types of Opportunism following previous Opportunism

<table>
<thead>
<tr>
<th>1st Transition</th>
<th>2nd Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Analogy</td>
</tr>
<tr>
<td>Analogy</td>
<td>1</td>
</tr>
<tr>
<td>Bottom Up</td>
<td>0</td>
</tr>
<tr>
<td>Debugging</td>
<td>1</td>
</tr>
<tr>
<td>Easy Goal</td>
<td>0</td>
</tr>
<tr>
<td>Jump Back</td>
<td>3</td>
</tr>
<tr>
<td>Parent not Coded</td>
<td>2</td>
</tr>
<tr>
<td>Post-requisite</td>
<td>0</td>
</tr>
<tr>
<td>Pre-requisite</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
</tr>
</tbody>
</table>

From Table 5.9 there are three transition pairs whose frequency seems to be significantly higher than what would be expected if all pairings were equally possible. These are Debugging-to-Debugging, Prerequisite-to-Prerequisite, and Prerequisite-to-Jump Back.
5.4.5.1. The Debugging-Debugging Pairing

It has been explained in a previous section that when a programmer tries to discover and amend bugs in a solution, it is not uncommon for the programmer to discover more than one bug.

As many bugs are often discovered in quick succession, the programmer will try to correct each bug in succession. This leads to Debugging Transitions to be mainly followed by another Opportunistic Transition for the purpose of debugging.

5.4.5.2. The Prerequisite-Prerequisite Pairing

Although Prerequisite Transitions often occur outside Clusters, there are instances where Prerequisites Transitions do occur in Clusters. In the cases where a Prerequisite Transition occurs in a Cluster, it is frequently followed by another Prerequisite Transition.

The Clusters that contain Prerequisite Transitions are the ones concerned with changing a solution to a problem. When a solution is changed, it is necessary to also change the prerequisite supporting components. As a solution can have many prerequisite nodes, it follows that changing a solution can involve many Prerequisite Transitions. Therefore it is not surprising that when a Prerequisite Transition occurs in a Cluster it is often followed by a similar transition.

5.4.5.3. The Prerequisite-Jump Back Pairing

Once the designer has added all the identified prerequisites of a solution, the designer can not make a transition that conforms to a global control strategy. Therefore the designers have to select Opportunistically the next node to describe.

The designers commonly decide to return to the original solution they were trying to complete before they deviated to produce the necessary prerequisites for the original solution, especially as the original solution was their concern when they made the Prerequisite Transitions.

5.5. Discussion

The hypothesis of the C study was that C programmers would employ a mixture of Breadth-First and Depth-First global control strategies. However the results of the study indicate that C programmers use a mixture of Children-First and Depth-First global control strategies. This behaviour is similar to Prolog programmers, although the C programmers made more Opportunistic Transitions than the Prolog programmers.

The results can be explained that the C programmers solved the problem of C’s openness not by using the Breadth-First strategy, but avoid the problem by using structured heuristic strategies. The problem can be avoided if the programmer designs the C modules to be closed, by making all variables local (i.e. the scope of each variable is the module in which it is declared), and by passing data between modules only via
parameters. As the C programmers constrained their use of the language, they made the main advantage of the Breadth-First strategy obsolete. Consequently, C programmers could use the less cognitive intense strategies of Children-First, without the solution to the problem being degraded.

The higher frequency of Opportunistic Transitions made by C programmers than made by Prolog programmers, can be accounted for by the different nature of the two languages. In C there are more chances for errors to appear, and also C solutions require more prerequisites than solutions written in Prolog. These two factors will respectively cause an increase in the level of Debugging Transitions, and cause an increase in the level of Prerequisite Transitions.

When the Opportunism of C and Prolog were compared it was apparent that the language affected the frequency with which some types of Opportunistic Transitions occurred. This leads to the question of what else could cause the frequency of types of Opportunistic Transitions to vary.

5.5.1. Factors Effecting the Frequency of Opportunistic Transitions

The first type 'Parent not Coded' is probably a result of the incompleteness of the verbal protocols and/or incompleteness of the program text (i.e. a sub-goal has been recognised but there is no indication that its parent has been recognised). Just because there is no indication that the parent has been recognised does not mean that it has not. The more verbose the programming language, then the more likely it is for the programming text to be more complete, and thus reduce the incident of PNC Transitions.

The frequency of the second type of transition, Jump Back, is a function of the frequency of the other Opportunistic Transitions. The occurrence of Analogy Transitions is dependent (1) on the software problem; (2) the solution chosen by the programmer to solve the problem; (3) the ability of the programmer to recognise analogies; and (4) the cost of applying analogies. The software tools used by software engineers can not affect the first two factors, but they can provide help with the last two factors. Software tools that support analogies, perhaps by providing a well-organised library of code, are liable to promote the Opportunistic Transitions of Analogy.

As has already been mentioned the occurrence of the transition of debugging is determined by the number of errors in the solution. Errors are a result of (1) the programmer's lack of knowledge; (2) the difficulty of the problem; (3) the complexity of the programming language; and (4) the complexity of the editor. There are many ways in which a programming language may be complex (syntax, semantics, and pragmatics). The complexity of syntax is a relative minor problem, especially given that programming languages have a restrictive syntax.
A more real problem is the complexity of semantics, and the possibility for ambiguity. Although each instruction in a language has well-defined functions, some of the functionality may be obscure and the cause of ambiguity. An example of a problem with semantics of C is the applying of operators on data types for which the operator was not designed.

The final dimension of complexity of a language is pragmatics, which is how the language is used. Pragmatics involves idiosyncratic use of the language, the programming culture, but also the semantics of the language. The more simplistic the semantics of a language is, then the more complex the pragmatics becomes. This is because with a simple semantics, the solution to a problem must be more elaborate. A single Prolog instruction can achieve the same results that would require many instructions from an assembly language.

The software environment can also affect the occurrence of Prerequisite and Post-requisite Transitions. The frequency of these two transitions is a function a solution requiring supporting solutions. The main source of Requisite Transitions is the use of data, and the viscosity of the language. For example if the programmer makes a change to a data structure in an Object Oriented language, then the programmer is liable to make only a few changes, so there will be few Requisite Transitions. As a counter-example, changes in a data structure in C will possibly involve changing all the operations performed on that data structure.

5.5.2. The Nature of the Problem Task

The problem task given to the designers in this study can principally be characterised as a data flow problem. In that the task is to receive data, manipulate the data, and output the results. Despite this, the programmers constructed the solution by primarily following control flow links in either a Depth-First or a Children-First manner. Of the remaining 23% of transitions, only the Requisite Transitions can be viewed as programmers following data links.

However it must be noted that data flow and control flow of the solution frequently follow the same path, especially if the solution is produced using structured programming techniques. Furthermore, all problems are a mixture of the control flow and data flow concerns, but some problems are predominately particularly worried with one of the concerns.

Whether a problem is one of data flow, or one of control flow, the resulting behaviour is likely to differ very little. Nevertheless, if the programmers were presented with a control flow problem rather than a data flow problem, then the frequency with which Requisite Transitions occur will probably be diminished. Consequently the occurrence of Jump Back Transitions will also diminish, and the overall percentage of conforming to a global control strategy will increase.
5.5.3. Advantages of the Breadth-First Control Strategy

The theorists of software engineering have prescribed that the optimum means for producing a well-designed solution is to use the Breadth-First global control strategy. Despite the proclamation of the theorist, the practitioners employ a mixture of the Children-First and Depth-First global control strategy.

This disparity is probably resultant from the theorists needing not to concern themselves with the cognitive costs of using a methodology (i.e. the amount of cognitive resources that need to be used to perform an action), while cognitive costs have strong persuasive effect on the practitioners as there is a limit to how much cognitive resources they can devoted to the activity. The cognitive cost is an extremely strong motivating factor for choice, given that many of the advantages of the Breadth-First strategy are covered by using the Children-First strategy in conjunction with structured programming techniques.

The advantage of the Breadth-First strategy is that the whole of each level of the solution is constructed before any part of the next level of the solution is constructed. If a section of the solution is found to be problematic, an alternative can be found that may involve alternatives to other parts of the solution also having to be found. Although the Children-First strategy is able to discover the same problems, the Breadth-First enables the problems to be discovered earlier on before the programmer becomes too committed to a solution.

The quality of the solution produced by the designers could be improved if the designer could be induced to use the Breadth-First strategy. This may be achieved by reducing the associated cognitive cost of the Breadth-First strategy. Software tools that perform some of the cognitive tasks used in the Breadth-First strategy could provide the help needed by programmers to persuade them to adopt the Breadth-First global control strategy.
Chapter 6
Local Control Strategies
Contents of Chapter

The previous chapters have examined the choice of global control strategies. This chapter focuses on the use of local control strategies that supplement global control strategies. Global control strategies specify a range of sub-goals that can be pursued. The local control strategy selects to pursue sub-goal from the specified range. There are a number of possible local control strategies. This chapter reports an experiment that observes the order in which programmers pursue a range of sub-goals. Each proposed local control strategy expects the programmers to pursue the sub-goals in a particular order. The expected behaviour is compared with the observed behaviour. The results indicate that the programmers select to pursue sub-goals according to either the order in which the sub-goals appear, or the order of dependency (from least to most) on the other sub-goals. The verbal reports tentatively suggest that the local control strategy is to pursue sub-goals in the order that they appear in the program text.

The chapter also discusses (1) whether text order determines order of dependency; (2) possible local control strategies used for more complex problems; and (3) whether behaviour that appears to conform to structured control strategies is Opportunistic.
6. Local Control Strategies

6.1. Introduction

In the previous Chapters, the global control strategies used by experienced programmers were examined. It was found that approximately 80% of transitions conformed to either the Depth-First or the Children-First global control strategies. However, the global control strategies only partly explain the order in which sub-problems are selected. Each global control strategy identifies a set of sub-goals at each stage of composition, the programmer can select to follow any one of the sub-goals from the set and still adhere to the adopted global control strategy.

For example, the Children-First strategy specifies that the programmer completes a goal, by first recognising each of its sub-goals. The programmer can then select any one of these sub-goals to complete, and after completing that sub-goal the programmer selects another sub-goal to complete. This continues until all the sub-goals have been completed, which entails that the goal itself has also been completed. Similarly, the Breadth-First global control strategy specifies that the programmer can select any sub-goal at a particular level to describe. After which, the programmer can select any other goal at the same level to describe.

Matters are somewhat different for programmers who use the Depth-First global control strategy. The Depth-First strategy details that programmers will start to describe a sub-goal immediately after that sub-goal has been recognised. The Depth-First strategy does not provide programmers with a set of sub-goals from which the programmer can select a sub-goal to describe. The Depth-First strategy provides the programmers with only one sub-goal to pursue. In essence, when the Depth-First strategy is guiding the programmer's construction order, guidance is determined purely by the locality of the sub-goal that the programmer has recognised.

6.1.1. Local Control Strategy is Needed to Supplement Global Control Strategies

Both the Children-First and the Breadth-First global control strategies only identify a set of sub-goals from which the programmer can choose a sub-goal to describe. To strengthen the explanatory power of the model of programming behaviour it is necessary to be able to also predict which sub-goal the programmer will choose, from any set of choices. In short, the local control strategy used by programmers needs to be identified.

6.1.2. Opportunism and Local Control Strategies

Opportunism can be viewed as the programmer using a local control strategy in absence of a global control strategy, since the purpose of both local control strategies and
Local Control Strategies

Opportunistic approaches is to select a sub-goal to describe from a pool of recognised sub-goals that have not yet been described. Local control strategies occur in the context of a global control strategy, whereas Opportunism occurs independent of a global control strategy.

If the programmers are performing in an Opportunistic manner, then they can choose to describe any sub-goal that has been recognised. Global control strategies reduce the options of sub-goals that can be chosen, by comparing the relative position of the sub-goals to the current selected sub-goal. For a programmer behaving Opportunistically, the relative position of the sub-goals is not sufficient grounds for considering a sub-goal as a non-valid choice. In Opportunism, the sub-goal's position may be reason for selecting the sub-goal, but it is never a reason for dismissing the sub-goal.

6.1.3. Possible Local Control Strategies

The study of local control strategies is as an important question for Opportunism as it is for conforming to a global control strategy. Unfortunately the data collected in the previous studies cannot provide answers to the question of which local control strategies are used by programmers. This is because the previous problems rarely required the participants to select from more than two conforming sub-goals. In the situation where the programmers did have a choice, the measurements of the choices are not accurate enough to test the hypotheses concerning the choice of local control strategies.

The studies that have identified programmers behaving Opportunistically have also identified possible factors that determine the selection of sub-goals. These factors may also determine local control strategies, as local control strategies have a similar purpose to Opportunism.

The number of potential factors for guiding choice is large, however the study reported in this chapter concentrates on the three most likely factors. The first potential factor is analogy. Visser (1990) proposed that if the programmer identifies a sub-goal that has similarities to a sub-goal that the programmer has already completed, then the programmer will select the newly identified sub-goal in order to make use of the analogy.

There are two levels of analogy (a) the specification level, and (b) the solution level. Two problems may be similar in their ends, and/or in their means to achieve its ends. As the programmer is interested in constructing the solution to a problem, then it follows that it is the second type of analogy that concerns the programmer. But as the programmers have not yet constructed the solution, they cannot know if the problem to be completed is analogous at the solution level to a completed problem. The programmer therefore has to identify the possibility for analogies at the solution level by
looking for analogies at the specification level. The programmer is liable to miss some analogies at the selection stage, but the ones that he has identified at the specification level are with high probability analogies at the solution level.

The second potential factor is the difficulty of the sub-goals. For example, the programmers may select the easiest sub-goal. It is possible that by describing a sub-goal, all the other sub-goals yet to be described will become easier. This is because, by describing a sub-goal, the ambiguity of the problem is reduced, perhaps providing pointers to the solutions of the other sub-problems, and thus simplifying the completion of the other sub-goals. Therefore describing the sub-goals in the order of Easiest-first may be the easiest method for describing all of the sub-goals.

The third potential factor is dependence of the sub-goals (Ball & Ormerod, 1995). Programmers who use this factor will select the sub-goal that is not dependent on any of the sub-goal not yet described, although the selected sub-goal may be dependent on the sub-goals that have been described. The advantage of this factor is that software tools can be used to test all completed parts of the overall solution at any stage. Unfortunately the factor of dependence can not be isolated from the text order in which the sub-goals are mentioned. This is because text order is used by programming languages to signify dependence. The data used by a sub-goal is dependent on data that preceding sub-goals have produced. In addition the execution of a sub-goal is dependent on the successful execution of the proceeding sub-goals.

In a text based programming language, data dependency and control dependency can not be isolated from the order of representation. Even in a visual programming language, data and control dependency can not readily be isolated from order of appearance, as dependency is typically determined by the order in which icons are connected to each other.

6.1.4. Hypothesis

Of the three potential factors, only dependence of goals has any overall advantage in helping the designer to achieve a solution that satisfies the specifications. The use of analogies may be advantageous, but the advantage can be used at any time, and not necessarily as soon as the analogy has been discovered. The only advantage of analogy that is dependent on time (i.e. when it is tackled) is retaining knowledge of the analogy. The greater the distance of time between describing the two similar solutions, then the more information the programmer will have to re-parse from the first similar solution to be used for the second similar solution (Gilmore & Green, 1987; 1988).

The idea that describing the easiest goal first will be the easiest method of describing all of the goal is also questionable. The false assumption is that describing the easiest sub-goal will make all the other sub-goals easier than the description of any other sub-goal would. However, programmers do not need to make easy sub-goals any easier,
Local Control Strategies

but they may need to make difficult sub-goals easier to describe. So although there is no overall advantage of selecting goals by their ease, the programmer may have concerns about individual difficult sub-goals that would lead them to select easier sub-goals in order to provide a framework for the more difficult sub-goals.

The dependence factor however does provide an overall advantage, by enabling the solution to be tested and corrected at each stage of completion. This prevents errors being replicated and compounded through the solution. The errors are weeded out before being allowed to spread to all subsequent described sub-goals.

As the dependence factor is the only one that has an overall advantage rather than temporary advantages, it is hypothesised that programmers will select sub-goals on the bases of the order in which they appear in the program text, since, text-order determines data and control dependencies.

6.2. Method

6.2.1. Design

The participants in this study were presented with a problem that contained four functions that required completion. The independent variable was the order in which the four functions were arranged in the program text. The order of the four functions presented to each participant is shown in Table 6.1. The dependent variable was the comparison of the order in which the programmers completed the four functions against the orders predicted by each of the three potential factors as shown in Table 6.2. Each participant was randomly assigned to one level of the independent variable of presentation order (i.e. the order in which sub-goals are presented to each participant is different).

6.2.2. Participants

Eight participants took part in the experiment. All of the participants were students at Loughborough University or employees of Loughborough University. The programming experience of the participants ranged from two to ten years. The mean length of experience was five years, and the standard deviation of experience was 2.45 years.

All of the programmers knew either Pascal, Modula-2, C or BASIC. Some of the programmers knew several of these languages, plus other additional languages. The participants with the greatest length of experience also had experience of the most programming languages.

6.2.3. Materials

All the participants were presented with a standard set of instructions (Appendix G.1.), which informed them that they would be presented with an incomplete software
solution. The instructions asked them to complete the four unsolved functions that are specified in the program text. In addition to the instructions the participants were presented with a simple program written in pseudo code that contained references to four functions that had not been written, and which the participants were required to complete. The order in which the four functions were mentioned in the program text was varied for each participant.

The program was presented in a window of the text editor TextRec, in which the programmers were required to complete the solution to all four functions. The same four functions were presented to each participant. The four functions were selected on the basis that each should have a different level of difficulty from the other functions, and that there is an analogy between the second easiest function and the hardest function. The purpose of this selection criterion was to be able to identify which strategy was used by each programmer to select the next sub-goal to describe. An example of an incomplete program presented to a participant is shown in Appendix G.2. The four incomplete functions, in the order of ease, were:

- the length of the longest time when the traffic survey did not register any vehicles.
- finished time in 12 hour clock of a traffic survey.
- the most common number of vehicles in a 1 minute period.
- the start time in 24 hour clock of the busiest 60 minutes of traffic.

The participants were also presented with a Questionnaire (Appendix G.3.) which asked for details of their expertise. The questionnaire also asked the participants to list the four functions in order of their perception of difficulty, starting with the function that they found the easiest.

6.2.4. Procedure

The participants were initially presented with a set of instructions (Appendix G.1.) that explained the procedure of the experiment. The participants were encouraged to ask any queries that they had.

The participants were asked to complete the four functions marked with a # in the program text that they were presented with. The programmers were required to use the text editor TextRec to produce solutions to the four functions. The programmers could write the solutions in pseudo-code or any language of their choice. The text editor, TextRec, recorded all human-computer interactions, as well as the final solutions to the four sub-problems.

While the participants were attempting to solve the four sub-problems, they were asked to verbalise all their thoughts. The participants' verbal reports were recorded onto audio tapes. If at any point the subjects became silent for more than one minute, they were prompted to continue speaking.
The programmers were informed that the problems could be solved in approximately one hour. The participants took between 45 and 65 minutes to produce their solutions. Once the programmers had completed the problems, they were asked to fill in a questionnaire (Appendix G.3.), which asked for details of the participant’s experience, and to rank the sub-problems in order of difficulty.

6.3. Results

6.3.1. Measurement of Local Control Strategies

The order in which the sub-problems were presented to the subjects was different for each participant. Each potential local control strategy provided a different prediction on the order in which the sub-problems would be pursued by each participant. Thus, the observed behaviour was compared against the prediction of the expected behaviour for each local control strategy.

Some of the local control strategies relied on the difficulty with which the participants perceived the sub-problems. The sub-problems presented to the participants were designed to have a certain level of relative difficulty. However, in the analysis of the data, the individuals’ reported difficulty of the sub-problems was used to determine the prediction of expected behaviour of the various local control strategies.

Table 6.1: Presentation and Difficulty of Sub-Problems

<table>
<thead>
<tr>
<th>Participants</th>
<th>Order of Presentation.</th>
<th>Order of Report Difficulty (Easiest first)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>dcba</td>
<td>abcd</td>
</tr>
<tr>
<td>C2</td>
<td>adcb</td>
<td>bdac</td>
</tr>
<tr>
<td>C3</td>
<td>bacd</td>
<td>badc</td>
</tr>
<tr>
<td>C4</td>
<td>badc</td>
<td>abdc</td>
</tr>
<tr>
<td>C5</td>
<td>cbad</td>
<td>b—</td>
</tr>
<tr>
<td>C6</td>
<td>cdab</td>
<td>bacd</td>
</tr>
<tr>
<td>C7</td>
<td>abcd</td>
<td>bacd</td>
</tr>
<tr>
<td>C8</td>
<td>dacb</td>
<td>bacd</td>
</tr>
</tbody>
</table>

Table 6.1 lists the sub-problems for each participant by (1) the order in which the sub-problems appeared in the program text for the participant, and (2) the order in which the participant reported difficulty of the sub-problems. Each sub-problem is identified by a letter, the meaning of each letter is:

a. length of the longest time when the traffic survey did not register any vehicles.
Local Control Strategies

b. finished time in 12 hour clock of a traffic survey.
c. most common number of vehicles in a 1 minute period.
d. start time in 24 hour clock of the busiest 60 minutes of traffic.

From the data in Table 6.1 it is possibly to derive prediction for the various proposed local control strategies. The participant C5 was unable to rank the sub-problems in order of difficulty as he found the functions a, c and d to be equally difficult, but he did find b to be the most difficult sub-problem.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Observed</th>
<th>Least Dependent First</th>
<th>Easy First</th>
<th>Difficult First</th>
<th>Analogies and Least Dependent First</th>
<th>Analogies and Easy First</th>
<th>Analogies and Difficult First</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>dcba</td>
<td>dcba</td>
<td>abcd</td>
<td>dcba</td>
<td>dbca</td>
<td>abdc</td>
<td>dbca</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.50)</td>
<td>(0.96)</td>
<td>(0.82)</td>
<td>(0.00)</td>
<td>(0.50)</td>
</tr>
<tr>
<td>C2</td>
<td>adcb</td>
<td>adcb</td>
<td>bcde</td>
<td>cdab</td>
<td>adbc</td>
<td>bcde</td>
<td>cdba</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.00)</td>
<td>(0.82)</td>
<td>(0.82)</td>
<td>(0.50)</td>
<td>(0.50)</td>
</tr>
<tr>
<td>C3</td>
<td>bade</td>
<td>bade</td>
<td>bcde</td>
<td>cdab</td>
<td>bdac</td>
<td>bdac</td>
<td>cdbe</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.82)</td>
<td>(0.00)</td>
<td>(0.50)</td>
<td>(0.50)</td>
<td>(0.50)</td>
</tr>
<tr>
<td>C4</td>
<td>bcde</td>
<td>bcde</td>
<td>adbc</td>
<td>cbde</td>
<td>bdc</td>
<td>adbc</td>
<td>cbda</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.50)</td>
<td>(0.50)</td>
<td>(0.00)</td>
<td>(0.50)</td>
<td>(0.50)</td>
</tr>
<tr>
<td>C5</td>
<td>cbad</td>
<td>cbad</td>
<td>b—</td>
<td>—b</td>
<td>cbda</td>
<td>bd—</td>
<td>db—</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(n/a)</td>
<td>(n/a)</td>
<td>(n/a)</td>
<td>(n/a)</td>
<td>(n/a)</td>
</tr>
<tr>
<td>C6</td>
<td>cdab</td>
<td>cdab</td>
<td>bcde</td>
<td>dcab</td>
<td>cdbe</td>
<td>bdac</td>
<td>dbca</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.00)</td>
<td>(0.82)</td>
<td>(0.82)</td>
<td>(0.50)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>C7</td>
<td>abcd</td>
<td>abcd</td>
<td>bade</td>
<td>cdab</td>
<td>abdc</td>
<td>bdac</td>
<td>cdbe</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.50)</td>
<td>(0.50)</td>
<td>(0.82)</td>
<td>(0.00)</td>
<td>(0.00)</td>
</tr>
<tr>
<td>C8</td>
<td>dacb</td>
<td>dacb</td>
<td>bcde</td>
<td>dcab</td>
<td>dbac</td>
<td>bdac</td>
<td>dbca</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.96)</td>
<td>(0.82)</td>
<td>(0.82)</td>
<td>(0.50)</td>
<td>(0.00)</td>
<td>(0.82)</td>
</tr>
</tbody>
</table>

Mean of Correlation | (0.96) | (0.45) | (0.63) | (0.73) | (0.29) | (0.41) |

Table 6.2 outlines the predictions of expected behaviour for each proposed local control strategy and for each participant. For the proposed expected behaviour where analogies is used in conjunction with another local control strategy, it is assumed that analogies is the principal strategy, and the other local control strategy is only used when there are no analogies. Table 6.2 also shows the correlation between observed behaviour and
Local Control Strategies

expected behaviour which are calculated using Kendall's τ. The scores for Kendall’s τ and the method of calculating τ are shown in Section J.1.

The data in Table 6.2 indicates that programmers choose the next sub-problem to complete on the basis of the first sub-problem to appear in the program text which has not been described. A possible explanation for the programmers' behaviour is that by selecting sub-problems on this basis they are selecting the sub-problem with the least dependence on the other incomplete sub-problems.

6.3.2. Verbal Reports

To supplement the quantitative data, verbal reports were collected. The verbal reports by themselves only provide hints of factors used to determine the selection of one of the four functions to describe. The only comments that the participants made when selecting one of the four functions were of the type 'Which is the next function?', or 'Now for the next function'. From the word 'next' it can be assumed that they are selecting the functions on the basis of some ranking order of the functions. However, the factors determining the order of selection of the sub-problem still have to be discovered. This question will have to be answered with the quantitative data, since the verbal reports never indicated what local control strategy the programmers used.

The participants often asked themselves which of the remaining problems they were to complete next. It would appear that the programmers did not rank all the sub-problems, and then use this ranking to make all subsequent choices. If the programmers did rank all the sub-problems when they made a choice, then they have forgotten or ignored this ranking on the next occasion when they had to make a choice. Alternatively, the programmers may only have found the sub-problem that would have appear first in the ranking order, and left the remaining sub-problems without an order of rank.

All the various possible methods for selecting a sub-problem require the comparison of each remaining sub-problem against each other on the basis of some factor. The exception to this is dependence, as the first uncompleted sub-problem to appear in the program text, is also the first uncompleted sub-problem not to be dependent on any of the other remaining uncompleted sub-problem.

In the verbal reports the participants rarely mentioned any of the sub-problems which they can select other than the one which they did select. This would suggest that the programmers did not compare the sub-problems with each other. This behaviour is compatible with selecting the sub-problems on the basis of dependence.

Needless to say there are alternative models of behaviour that can fit with the observed verbal reports. For example the programmer's method of selecting sub-goals may be a procedural activity. The processes of procedural behaviour are not readily verbalised, and this could account for the lack of mention of comparisons being made.
6.3.3. Measurement of Difficulty

Quantitative measurements of completing the four functions were collected in addition to the self reports of function's difficulty. The purpose the quantitative measurements was to verify the participants' opinion of difficulty. The three quantitative measurements collected were (1) the complexity of the solutions; (2) the time taken to produce the solutions; and (3) the percentage of interaction that the editor associated with each function.

Table 6.3: Self Reported Ease Compared Against Quantified Ease

<table>
<thead>
<tr>
<th>Self Reported Ease</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>31.00</td>
<td>85.14</td>
<td>94.14</td>
<td>97.86</td>
</tr>
<tr>
<td>Time Taken (min)</td>
<td>8.68 min</td>
<td>12.55 min</td>
<td>17.08 min</td>
<td>19.47 min</td>
</tr>
<tr>
<td>% of Interaction</td>
<td>14.52%</td>
<td>20.70%</td>
<td>31.03%</td>
<td>31.12%</td>
</tr>
</tbody>
</table>

The complexity of a solution is calculated by representing the solution as a hierarchy. The complexity of a solution is the sum of the complexity of its sub-goals plus one. Where a solution does not have any sub-goals then its complexity score is one. A simple method of arriving at the score of complexity is to sum the level at which each node occurs. The top node occurs at level one, its children at level two, their children at level three, etc.

The complexity of the solution, the time taken to produce the solution and the percentage of interaction devoted to the problem were measured for each participant and for each problem. Table 6.3 shows the mean scores of the problem ranked by ease of solving, while Tables 1.3–1.5 reports the scores for each participant. The quantitative measures of the solution match well with the self reported ease of problem. The easiest problems were less complex, completed in less time, and required few interactions, whereas the opposite is the case for the most difficult problems. The Page's L Trend test confirmed that these trends were significant at the 95% confidence level (c.f. Sections 1.3–1.5).

The time taken to complete each function was also compared against the order in which the functions were solved. Table 6.4 shows how much longer it took the programmers to complete a problem against the hypothesised time that all problems were completed with equal amount of time after taking into account difficulty of the sub-problems.
Table 6.1: Time Taken to Solve First-Fourth Sub-Tasks

<table>
<thead>
<tr>
<th>Order of Solution</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra Time Taken (%)</td>
<td>+10.56%</td>
<td>-9.48%</td>
<td>+1.07%</td>
<td>-0.77%</td>
</tr>
</tbody>
</table>

The first function took 10.56% longer to complete than would have been predicted if the order of solution had no influence on the time taken. The second function took 9.48% less time to complete than would have been predicted. With the last two functions closely matching the amount of time in which it would have been expected them to be completed. Section J.5. reports the extra time taken per sub-problem for each participant. Section J.5. uses the Page’s L Trend test to calculate whether the possible trend was significant ($k = 4, N = 8, L = 209$). At the 95% level it was found not to be significant, although it was very close. There were large variations in the extra time taken to complete the solution.

The probable explanation for the possible change of time taken against the order in which the functions are tackled is that information gathered by programmers during their attempt to solve their first problem is used in their attempt to solve the remaining problems. For example the input for each of the four functions is the same. Once the programmers have discovered the structure of the input for the first sub-problem they have tackled, then they need not to spend time discovering the structure of the input for the remaining function, since they have already discovered it.

Despite the accumulation of information this possibly does not prevent the programmers from taking longer to complete the third and final function than it did to complete the second function they tackled. Many of the programmers reported fatigue towards the end of the task, which may have caused them to slow down and to take longer to complete their solutions to the remaining problems.

6.4. Discussion

The qualitative verbal reports and the selection order data provide tentative support that programmers select a sub-problem from the pool of potential sub-problem by the first-come first-served strategy. This conforms to the hypothesis that programmers select the sub-goal which is the least dependent on the other sub-goals in the option pool. The quantitative data provide additional support for this strategy, while providing evidence that the other suggested local control strategies are not adhered to by the programmers.
6.4.1. **Is Local Control Strategy Determined by Dependence or Text Order?**

In this study it was assumed that programmers selected goals on the basis of the presentation order in the program text, as the selected goal would be the one least dependent on the other remaining goals to be described. However it is possible that programmers where selecting goal on presentation order purely because it was a simple local control strategy that consumes few cognitive resources, and they do not perceive an advantage in employing a more complex local control strategy. This would be especially true if the programmers employed structured programming techniques. In such situations the only two sources of dependencies that exists are (1) those defined by the parameters being pass to and fro between modules; and (2) the requirement that proceeding modules must succeed before subsequent modules can be processed. Neither of these dependencies affect the solution of each module. As dependence does not affect the solution of a module, there is little reason for the programmer to use this factor in selecting the next module to solve.

Another alternative explanation of the results is that the programmers were making detailed comparisons between the competing sub-goals, without using text order as the controlling factor. The verbal protocols suggest that this alternative explanation is incorrect, but as the verbal reports are not complete this alternative explanation can not be ruled out.

A further experiment is required where the text order and the dependence order of the modules differ. Potential possibilities are studies using the Prolog language or a visual programming language. In theory the clauses of a Prolog goal can appear in any order, although in practice the clauses are ordered by dependence, otherwise the goal may fall into an infinite loop. With some visual programming languages the sub-goals can appear in any order, with the dependencies indicated by lines pointing from one sub-goal to another.

6.4.2. **Local Control Strategy for Richer Problems**

In this experiment participants were required to complete four sub-problems in approximately 60 minutes. In comparison to the problems that programmers typically have to solve, this task is rather simple. The problems faced by programmers may be more complex is the following ways: (1) there are more sub-goals for the local control strategies to handle; (2) the sub-goals may be more difficult; and (3) there may be greater inter-dependencies.

If sections of a problem were to contain more sub-problems, this would make the task of comparing the sub-problems require more processing, except where the first-come first-served local control strategy is used. However, it is unlikely for a problem to contain many more than four sub-problems that need to be further decomposed. The
number of sub-problems that need to be decomposed is largely determined by the gap of abstraction between the problem and the sub-problems. The greater the gap of abstraction between a problem and its sub-problems, then the more difficult it becomes to determine whether all the sub-problems together achieve the aims of the problems. As programmers wish to produce correct solutions, they will limit the gap of abstraction between the problem and the sub-problem. This consequentially limits the number of sub-problems that a problem is decomposed into, and explains why most programs are hierarchical rather than flat.

If the sub-problems were made more difficult then the programmers may not be able to use the first-come first-serve strategy. This strategy may suggest to the programmers to complete a sub-problem which they do not feel capable of completing (the programmers may lack expertise, or they may be too tired, or the problem may be too ambiguous). With difficult problems, the programmer may have to use an Easiest-first local control strategy, or a first-come first-served strategy which skips a very difficult sub-problem.

The final way in which a problem may be more richer is if the sub-problems have greater inter-dependence, for example if one of the sub-problem is also a child of another sub-problem (programs are not strictly hierarchical, but a direct network). Even then the first sub-problem will be the least dependent, so a first-come first-served strategy will still probably be advantageous.

6.4.3. Is a Global Control Strategy Opportunistic?

In the introduction it was suggested that Opportunism is identical to local control strategies which are not supported by a global control strategy. That is to say that the purpose of Opportunism and local control strategy is the same. Their purpose is to select a sub-problem to describe or complete from a pool of potential choices. The difference is that local control strategies occur in the context of a global control strategy which restricts the set of choices on the basis of the sub-problems’ relationship to the current selected sub-problem. Whereas Opportunism occurs in a situation where the pool of choice has not been restricted.

The model that has been advanced so far is that programmers rely on global control strategies, except when a problem with the solution being produced is encountered, at which point the programmer Opportunistically rectifies the problem, before continuing with a global control strategy to construct the rest of the solution. Furthermore where the explanatory power of global control strategy is lacking, the local control strategies provide the prediction.

There is an alternative model which has the same explanatory power. This alternative model is that all behaviour is Opportunistic, and it is often opportunistic to move from one sub-problem to a closely related sub-problem which accounts for about
Local Control Strategies

80% of all transitions. In this opportunistic model programmers have a number of aims, and the programmer selects the sub-goal whose completion will resolve the aims more than the completion of any other sub-goal would. The main aim is to construct an error free solution that meets the requirements of the problem. To achieve this aim, the programmers' first concern is to (1) rectify any errors that have been spotted. The other concerns are (2) to produce components that compatible with each other; and (3) to produce the solution in a manner that facilitates the identification of errors.

The first concern accounts for divergent transitions from global control strategies. While the other two concerns would explain behaviour that is also explained by global and local control strategies. The second and third concerns would lead programmers to construct software by following control flow links in a forward fashion. The consequence is behaviour that overtly appears that the programmer is either following the Children-First or the Depth-First global control strategies, with a Least Dependence-first local control strategy. This behaviour would also entail that a solution at any time in its construction can be tested with software tools for errors in the solution.

In this alternative model global control strategies are seen as an artefact of Opportunism. Although the two models explain the same overt behaviour, they rely on different cognitive behaviour. The main cognitive difference between the two models is whether programmers deliberately reduce their choice to a few sub-goals (as they do with global control strategies) before selecting a sub-goal to pursue; or whether they are Opportunistically predisposed to pursue certain sub-goals (which happen to be the ones chosen by global control strategies) from the choose of all sub-goals.
Chapter 7

Using the Breadth-First Global Control Strategy
From the previous chapters it was found that programmers do not use the Breadth-First control strategy. This chapter examines the reasons why programmers do not use the Breadth-First control strategy and whether the Breadth-First control strategy promotes good solution as has been suggested by other researchers.

The hypothesis is that programmers use alternative control strategies as the Breadth-First control strategy is cognitively expensive, and that standard text editors exacerbate the cost of using the Breadth-First control strategy. The study in this experiment evaluated the occurrence of Breadth-First behaviour for three text editors. The first editor, was neutral to the control strategy employed by the programmers, the second editor supported programmers using the Breadth-First control strategy, and the third editor enforced use of the Breadth-First control strategy. The results of the experiment showed that even when programmers were given text editors that need not exacerbate the cost of the Breadth-First control strategy, programmers still chose not to use it when they were given the option.

However, the reason for not choosing the Breadth-First control strategy can not be because it is not as effective as alternative control strategies. The programmers who were forced to use the Breadth-First control strategy were more productive and they produced better solutions than programmers who used alternative control strategies.
7. Using the Breadth-First Global Control Strategy

7.1. Introduction

In the previous studies it has been observed that programmers do not employ a Breadth-First global control strategy to guide their construction of a software solution. Instead programmers employed a combination of the Depth-First and the Children-First global control strategies with occasional opportunistic deviations from these two top-down global control strategies.

Software design gurus have argued that programmers should employ a Breadth-First strategy to guarantee a well-defined solution to a software problem (Wirth, 1971). By definition experts are those who will produce the best solutions, so one would expect experts to employ the best strategy which is supposedly the Breadth-First strategy. Given that expert programmers do not use the Breadth-First strategy, the following questions need to be addressed: (1) why programmers do not use the Breadth-First global control strategy, and (2) whether the Breadth-First global control strategy is better for producing a well-defined solution than the alternative global control strategies which programmers do employ.

7.1.1. Parsing Required for Each Global Control Strategy

An important cognitive component of generating a software solution is the comprehension of the solution generated so far. For example, when programmers have described one part of the solution and move to a new section to describe, they have to comprehend the current state of the solution that impinges on design of the new section.

Different global control strategies may require differing amounts of parsing (i.e. acquiring information from external memory or long-term memory, and placing it in working memory) of solutions in order for the programmer to be able to describe the next immediate sub-problem. As software design is a complex cognitive task, the programmers may well choose to employ a global control strategy that lightens the cognitive cost of the parsing task. For example, programmers may prefer global control strategies that require a small amount of parsing over strategies that require a greater amount of parsing.

The primary situation where parsing is required, is where the programmer makes a transition from one part of the software problem space to another part. Intuitively, it appears that the Children-First global control strategy requires less parsing than the Breadth-First global control strategy. Most transitions in the Children-First global control strategy are principally between two sibling nodes, whereas a transition in the Breadth-First global control strategy can be between two peers (any two nodes of the
same generation). Although it is preferable for reducing the degree of parsing, that the peers are closely related, it is not always possible. The greater the distance between the two nodes, then the less in common the two nodes are likely to have, therefore the information needs of the programmer increases.

Below is a detailed explanation of why the Depth-First global control strategy requires the least amount of parsing, followed by the Children-First global control strategy, with the Breadth-First global control strategy requiring the most parsing.

7.1.1.1. Examination of Parsing Required for Each Global Control Strategy

Each global control strategy specifies which transitions from one node in the hierarchy to another node conform to the strategy. With each transition the designer must parse addition information that is necessary to describe the new node, but which was not used during the description of the previous node. By examining the nature of a transition it is possible to estimate the amount of parsing of additional information that is required. As the general nature of transition for each global control strategy is known, it is possible to calculate the overall amount of parsing required depending on the global control strategy used.

Parsing Required for the Children-First Global Control Strategy

The principle transition for describing a node in the Children-First global control strategy is to move from the current node being described to one of its offspring in order to describe that offspring. No parsing of additional information is required for this transition, since the information needed to describe the new node is either the same information used to produce the previous node; or information generated during the production of the previous node. The designers may have to do some minor parsing to refresh their memory. But as this is the case with all transitions for all global control strategies, this minor parsing can be eliminated from consideration. The only consideration of this argument is the major parsing of whole segments of code represented by a node in the hierarchy.

If a node has been described which has no offspring needing to be described, then the next transition is to a sibling of the node. Like the previous type of transition, this one requires no additional information to be parsed. All the information necessary for the sibling node, was also necessary to describe the previous node.

The final type of transition that can occur within the Children-First global control strategy is to move from the node just described to its nephew. If there does not exist a nephew that needs to be described then a nephew of the node's youngest ancestor will be selected to be described. In both cases the ancestors of the new node are also ancestors of the old node, except for the new node's parent. Therefore only the
Using the Breadth-First Global Control Strategy

information relating to the new node's parent needs to be parsed to supplement what the designer is already aware of.

In short, the principle transitions of the Children-First global control strategy require no major parsing. The only transitions that require any major parsing, require the parsing of information relating to only one node. At no stage is it necessary to parse two or more nodes.

Parsing Required for the Breadth-First Global Control Strategy

The Breadth-First global control strategy requires that the designer completes one level of the hierarchy before move down to the next level. This means that nearly all the transitions in a Breadth-First global control strategy are between peers.

It is reasonable to assume that all peer transitions are between the closest related peers as this would reduce the amount of parsing the programmer needs to perform. In the majority of cases this will be a transition between siblings. The information used to describe the previous node is the same as the information needed to describe the sibling. So in the majority of cases, sibling transitions, no major parsing is required.

However if there are no siblings left to describe, the transition must be between nth cousins, with the smallest possible nth. Nth cousins have n ancestors which they do not share with their reciprocal relation. So transitions between nth cousins require the parsing of information from the n nodes which were not relevant to the previous node. n will most frequently be one, then two, then three, etc. The maximum possible value of n is the number of ancestors the new node has minus one, since peers need not share any ancestors except for their first ancestor. If it is not assumed that transitions between peers are between the closest related peers possible, the amount of parsing required would increase.

Once all the nodes of the same generation have been described, the designer must move to the next level. This transition need not require any parsing provided that the transition is from parent to offspring. For this argument it will be assumed that this is the case. However moving from one level to the next is more likely to require a lot of parsing, as expert designers tend to work linearly forward from beginning to end. So when moving from one level to the next, the designer will probably be moving from the end of one level to the beginning to the next. The node at the end of level X only shares one ancestor with the node at the beginning of level X+1. The deeper the designer goes the greater the amount of parsing that is necessary when moving down a level.

To summarise, the majority transitions in Breadth-First global control strategy require the parsing of no extra nodes. The second most common transition is between first cousins that require the parsing of one node. The third most common transition is between second cousins that require the parsing of two nodes, etc.
Comparing Children-First against Breadth-First Global Control Strategy

The transitions in the Children-First global control strategy never require the parsing of more than one node, whereas the number of nodes that are required to be parsed by the Breadth-First global control strategy is only limited by the number of ancestors of the new node. Nevertheless this does not prove that the Children-First global control strategy requires less parsing than the Breadth-First global control strategy. The frequency of transitions which require no parsing may be higher in the Breadth-First global control strategy than in the Children-First global control strategy. The amount of parsing each strategy requires is the sum of the product of frequency and amount of parsing.

\[ \sum = f_i \times i \]

where \( f_i \) is the frequency of transitions that require \( i \) nodes to be parsed.

The only situation where a transition in the Breadth-First global control strategy does not require any parsing is when the designer is moving between siblings, or on the rare occasion of moving from parent to offspring. If the designer has finished a node which the Breadth-First global control strategy specifies that the designer moves to a sibling, what would the Children-First global control strategy specify? The Children-First global control strategy would direct the designer to move to an offspring if one exists which needs describing. If there is not one then the Children-First global control strategy would direct the designer to move to an offspring or a sibling.

If the designer has finished a node which the Breadth-First global control strategy specifies that the designer moves to a sibling, the Children-First global control strategy would also specify that the designer moves to a sibling.

So in all the situations where the transition specified by the Breadth-First global control strategy does not require any parsing, are also situations where the transition specified by the Children-First global control strategy does not require any parsing. This demonstrates that the frequency of non-parsing transitions cannot be higher for the Breadth-First global control strategy than for the Children-First global control strategy.

However in situations where the Children-First global control strategy specifies a non-parsing transition to an offspring or sibling, the Breadth-First global control strategy specifies at best a non-parsing transition to a sibling. But it is possible that there are no siblings that need describing, so the Breadth-First global control strategy may specify a transition to an nth cousin which requires some parsing. While there are no situations where the Breadth-First global control strategy can specify a non-parsing transition when the Children-First global control strategy does not, there are situations where the Children-First global control strategy specifies a non-parsing transition when the Breadth-First global control strategy does not. So the frequency of non-parsing
transitions is lower for the Breadth-First global control strategy than for the Children-First global control strategy.

This means that the frequency of parsing transitions is higher for Breadth-First global control strategy than for the Children-First global control strategy. In addition the amount of parsing when required for the Children-First global control strategy never exceed the parsing of one node. When parsing is required for the Breadth-First global control strategy the amount of parsing required can exceed the parsing of one node, and by definition can not be less than one node. As the frequency of parsing and the amount of parsing undertaken when it occurs is higher for Breadth-First than Children-First global control strategy it follows that the overall amount of parsing necessary for a Breadth-First global control strategy is higher than the amount necessary for a Children-First global control strategy.

Parsing Required for the Depth-First Global Control Strategy

The principle behind the Depth-First global control strategy is to describe and complete a node as soon as it has been recognised. The result of this principle of describing a node as soon as it has been recognised is never to finish describing a node until all of its descendants have been described. Therefore the designers are either moving down the hierarchy to describe the newly recognised node, or moving up the hierarchy to finish describing a node that the designers had originally started on when they were immediately interrupt by recognising an offspring which needed described.

This moving only up and down the hierarchy means that at no point does the designer move from one node to a new node, where the new node's ancestors are not either old node or ancestors of the old node. So for each transition the information (or part of it) gained from the old node and from the old node's ancestor is all the information needed for describing the new node. Therefore at no stage does the designer need to perform major parsing of a node.

Summary

It would appear that the strategy which the designers select will affect the amount of parsing they will perform. Consequently this will affect the cognitive workload of designing. The amount of major parsing required when using the Depth-First global control strategy is zero. On the other hand, the Breadth-First global control strategy requires the most amount of major parsing. The amount of parsing required by the Children-First global control strategy is between the two extremes.

7.1.2. Exacerbating the Parsing Problem

The parsing problem of the Breadth-First global control strategy is unavoidable. However, conventional software tools exacerbate the problem. In order to parse the relevant sections of the solution, it is first necessary to locate the sections to be parsed.
Conventional editors merely present an unfiltered view of the current solution, i.e. the order in which the programmer ordered the solution.

The order in which the programmers created the solution is not necessarily the optimal presentation to aid the programmer in mining for the information they require to describe the solution to the sub-problem they are currently tackling. Indeed for any control strategy a conventional presentation can not always be the optimal presentation for mining of the relevant information. However as the Breadth-First global control strategy suffers most from the parsing problem, the Breadth-First global control strategy is hindered most by any exacerbation of the parsing problem.

The optimal presentation for the mining of information is to present in a coherent manner only the sections of the solution that may contain useful information. For any one sub-part of a solution all the necessary information is located in the higher levels of the solution that that sub-part is a descendent of. For example node if a programmer is describing the node 1.3.2.4, then the information he may require is located in the description of node 1.3.2; node 1.3; and node 1. These higher level nodes contain information on the input and output of node 1.3.2.4, i.e. information on the node’s requirement. (The programmer may need additional information but any additional information will not be contained within the program text, but in some other text source.)

In order for programmers to be encouraged to use the Breadth-First global control strategy an essential requirement is not be exacerbate the parsing problem. Programmers at least need to use a software editor that filters the program text into a manageable comprehensive segment.

7.1.3. An Additional Problem with the Breadth-First Global Control Strategy

Besides the problem that conventional editors do not provide an ideal presentation of the software solution which affects the Breadth-First global control strategy most amongst top-down control strategies, the Breadth-First global control strategy has an additional problem. As a software problem becomes increasingly more complex, the solution is increasingly broken down into more components. With an increase in the number of components, it becomes more problematic to remember where each component fits into the overall solution.

For the Depth-First and the Children-First global control strategies this remembrance of position is not important. The programmer employing either of these two strategies only has to remember the components that are closely related to the current component under development. If the programmer forgets which components are closely related it is a simple task to recover this information. For programmers employing the Breadth-First global control strategy, it is necessary to determine more
Using the Breadth-First Global Control Strategy

distant relationships (i.e. the peers of current component). The component's peers include closely related components as well as more distant relations such as the nth cousin. With a complex problem it becomes more demanding to remember which components are peers of other components, and also if the programmer does forget whether a component is a peer of the current component it is difficult to determine if two components are peers when they are distantly related.

7.1.4. The Aims of the Study

To encourage programmers to employ the Breadth-First global control strategy it may be necessary to provide a software tool that automatically determines the relationship between components, thus easing the cognitive cost placed on the programmers who are using the Breadth-First global control strategy.

The aim of this study was to determine the requirements necessary to encourage programmers to employ a Breadth-First global control strategy, and secondly whether programmers who employ the Breadth-First global control strategy perform better than programmers who employed an alternative control strategy.

To achieve this aim three different text editors were produced. The editors differed by the support they gave to programmers to use the Breadth-First control strategy. The first editor gave no support for the Breadth-First control strategy. The second editor provided some support by indicating the order in which the solution should be decomposed. The final editor forced programmers to conform to a Breadth-First control strategy. Section 7.2.3. on Materials provides a detail description of the three different text editors.

7.1.5. Hypothesis

The hypothesis of this study is that programmers will employ a Breadth-First global control strategy when they use a helpful editor that (1) filters the view of the software solution, and (2) informs the programmer of the relationship between components. It is also hypothesised that solutions produced using the Breadth-First global control strategy are better than solutions produced using an alternative control strategy.

7.2. Method

7.2.1. Participants

Thirteen programmers with knowledge of C participated in this study. The participants were primarily undergraduate Computer Science students at either Loughborough University or Lancaster University. One participant was an electrical engineering student at Lancaster University. Two of the participants were postgraduate computer science students at Lancaster University.
Using the Breadth-First Global Control Strategy

All of the participants had at least two years experience of designing and implementing programs. On average the participants had just over seven years of programming experience. The participants all had basic knowledge of the computer programming language C.

7.2.2. Task
The task presented to the participants was chosen on the basis (1) that it was possible to complete the task in 60 minutes; and (2) that the solution to the task would have a hierarchical structure that was complex enough that it was possible to distinguish which global control strategy each participant used.

The task presented to the programmers contained five sub-tasks. These were to (1) calculate the average flow of traffic from an array of road use; (2) calculate the start index of the longest period of time when the road was not in use from an array of road use; (3) convert the start index into a start time of the twelve hour clock format; (4) output the average flow of traffic; and (5) output the start time of the longest waiting period. The description of the program to be written is given in Appendix H.6., this was presented to the participants.

An additional characteristic of the task was that participants were only allowed to block code into procedures. This meant that the structures within the procedures were flat, and that the hierarchical structure of the solution was described solely by the relationship between the procedures. For example, if the participants wanted to produce a while-do loop, the body of the loop had to be described in a separate procedure. In C, this meant the curly brackets could only be used to delineate a procedure, and nothing else.

This restriction not only caused the structure within a procedure to be flat, thus enabling software tools to impose on programmers the Breadth-First global control strategy, but it also caused the solution to consist of many more procedures than it would otherwise have done. This enabled the solution to be complex enough to determine what global control strategy each participants used.

7.2.3. Material
The main aspect of the study is the examination of the editor's influence on the global control strategy used by programmers; and examination of the usefulness of the various strategies which the programmers may use. The study employs three different editors to investigate these questions. The three editors were designed to be identical to each other except in one aspect. Any difference discovered in the strategy of programmer using different editors could be identified as a result of the editors used.

The editors were based on TextRec, the editor used in the previous studies, and which in turn is based on the text editor TeachText. TextRec is a simple editor that
Using the Breadth-First Global Control Strategy

provides only the simple functions of editing. TextRec also records all the keystrokes produced by the user.

By comparing the requirements necessary to conform to the various top-down control strategies, it is apparent that the Breadth-First global control strategy requires more parsing of program text than any other top-down control strategy. With a standard text editor, the ability to parse the information necessary for the Breadth-First global control strategy may be hindered as the required information is scattered in the program text. Thus the programmer has to mine for the data as well as parse it. As a consequence a standard text editor is not neutral with regard to the choice of control strategy. The standard text editor hinders programmers conforming to the Breadth-First global control strategy.

A neutral text editor would be one that mines for the information that the programmer may require to parse while using any top-down control-strategy. The information that the programmers require is the constraints on the current module that the programmer is trying to produce. The constraints of a module are dependent on its ancestral modules, that is all higher level modules that encompass the functionality of the current module. So a neutral editor would only display the current state of the module being worked upon and its ancestral modules.

The three text editors used in this study all mined for the necessary program text, and displayed it as a collective whole, rather than displaying the whole program text with the necessary parts scattered throughout it. For example, if the programmers were working on node 1.3.2.4, the solutions to nodes 1, 1.3, 1.3.2, and the partial solution of node 1.3.2.4 will be displayed, while the solutions to all the other nodes are not shown.

In order for the editors to know which information to mine for, the programmers need to inform the editor which module they want to develop further. In the three editors used this was achieved by the programmers selecting the module they want to develop from a menu called 'Move'. The Move menu lists in alphabetical order all the modules that are mentioned in the program text. The editors were configured to identify modules by searching for function calls in the C programming language. Function calls to library routines were ignored as the programmers do not need to develop those functions. All the remaining functions called (i.e. those to be described by the programmer) were listed in the move menu. For example from the following program text
Using the Breadth-First Global Control Strategy

```c
main()
{
    function1();
    function2();
    printf('hi')
}
```

the text editors would list the functions `function1`, `function2`, and `main` in the Move menu.

The three editors differed by which functions the editor allowed the programmer to choose, and by which functions the editor encouraged the programmer to choose. The neutral editor, `TextRec+n`, allowed the programmer to select any module to develop, and the editor did not encourage the programmer to select any particular module.

The second editor, `TextRec+s`, was supportive of the Breadth-First global control strategy, it also allowed the programmer to select any module, but the editor underlined the modules whose selection would conform to the Breadth-First global control strategy.

The final editor `TextRec+e`, enforces the programmer to conform to the Breadth-First global control strategy. The programmer could only select the modules that were specified by the Breadth-First global control strategy. The editor allowed the programmers to move between any modules on the same level, or to any module on the immediate lower level, but once the programmers had moved down a level they could not move back up a level. `TextRec+e` warns the programmers each time they choose to move down a level and they are presented with a choice. The choice was to proceed or stay on the current level. The warning was that their solutions to modules of the current level will not be editable if the programmer moves down a level.

### 7.2.4. Procedure

At the beginning of each experiment, the participants were presented with information that outlined the nature of the task that they will be presented. The information they were presented with also included a description of the Breadth-First global control strategy and a description of the editor they were asked to use (c.f. Appendix H.1-H.5).

When the participants were happy that they understood the information presented to them, the experimenter handed them the specification of the C program they were to produce.
The participants were asked to think aloud as they produced their solutions. They were given 60 minutes to complete the program. If after 60 minutes they had not manage to complete the program they were asked to stop.

The participants had to produce their solution using one of the three text-editors. There were three text editors (1) an editor that supported each top-down control strategy equally; (2) an editor that supported the Breadth-First global control strategy; and (3) an editor that forced programmers to adopt the Breadth-First global control strategy. Each participant was randomly assigned to use one of these three editors. The supportive editor and the enforcing editor was each used by four participants. The neutral editor was used by 5 participants.

7.3. Results and Discussion

7.3.1. Conforming to the Breadth-First Global Control Strategy

One question of this study is whether programmers would use a Breadth-First global control strategy when the software environment was either neutral to the various control strategies or when the environment was supportive of the Breadth-First global control strategy.

As the internal structure of the procedures was flat it is possible to measure conformance to the Breadth-First global control strategy by examining the order in which the programmers edit the various procedures. If the programmers were employing the Breadth-First global control strategy they would first edit all the procedures of the highest level, before editing all the procedures of the second highest level. The programmer would complete all procedures at one level of abstraction, before moving down to the next lower level of abstraction. Thus a programmer using the Breadth-First global control strategy would either move between two procedures at the same level of abstraction, or move down to a procedure at the next level. Any other movement between two procedures would not conform to the Breadth-First global control strategy.

Occasionally when the participants choose to move to another procedure to edit, they would choose the procedure they were already editing. In the following results these movements have been included, but they have not been assigned to either conforming or diverging from the Breadth-First global control strategy.

In the introduction it was hypothesised that programmers in the previous studies do not conform to the Breadth-First global control strategy for two reasons. These were (1) that the Breadth-First global control strategy required more parsing than either of the other two top-down control strategies, furthermore the conventional text editors exacerbated the cognitive cost of recovering the information; and (2) that it is difficult to
Using the Breadth-First Global Control Strategy

remember which procedures are at the same level of abstraction as the current procedure being editing.

The neutral editor was produced to remove the first reason for not using the Breadth-First global control strategy, while the supportive editor also removed the second reason for not using the Breadth-First global control strategy.

Table 7.1: Movement between Procedures

<table>
<thead>
<tr>
<th>Environment</th>
<th>Conforming</th>
<th>Diverging</th>
<th>Stationary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral Editor</td>
<td>45.81%</td>
<td>46.07%</td>
<td>8.12%</td>
</tr>
<tr>
<td>Supportive Editor</td>
<td>45.28%</td>
<td>48.05%</td>
<td>6.67%</td>
</tr>
<tr>
<td>Enforcing Editor</td>
<td>93.91%</td>
<td>0.00%</td>
<td>6.09%</td>
</tr>
</tbody>
</table>

As can be seen from the results in Table 7.1 (Table 1.11 which shows the individuals scores), reducing the first discriminating factor does not encourage the use of Breadth-First control strategy. When the second discriminating factor is also removed the still programmers avoided using the Breadth-First global control strategy as much as the programmers who used the neutral editor. The programmers in this study only used the Breadth-First global control strategy when they were forced to do so by the editor. It is clear that programmers were not willing to employ the Breadth-First global control strategy.

A clue to the reluctance of the programmers to employ the Breadth-First global control strategy can be found in the comments of the programmers who were forced to use the Breadth-First global control strategy.

Subject D4 made one value judgement that the Breadth-First global control strategy does not comply with a facility he expects from a control strategy, and he made a further comment about the difficulty of using the Breadth-First global control strategy. These comments were echoed by the other three programmers forced to use the Breadth-First global control strategy.

D4's first comment was “I like to know something works before going on.” The problem with the Breadth-First global control strategy is that the solution at any stage of construction can not be tested by the common method of compiling and running the program and comparing the program's actual behaviour against the program's required behaviour. To test any section of a solution by running it, it is necessary for that section of the solution to be described at all lower levels of abstraction. The Breadth-First global control strategy explicitly forbids the description of all lower levels of
Using the Breadth-First Global Control Strategy

abstraction until the final stages of constructing the solution. As the Breadth-First global control strategy in a conventional environment prevents a common method of testing the solution, the programmers may be avoiding the Breadth-First global control strategy in favour of other control strategies that enable this type of testing.

D4's second criticism of the Breadth-First global control strategy was "when you've done the second one, you go back to the first one and you've forgotten what [you've] done." Essentially, this is a verbalisation of the parsing problem. In the Breadth-First global control strategy the programmer may have to move from one section of the program to another which has no direct relevance to the previous section. When the programmer moves back to the previous section to describe it at a lower level of abstraction, the programmer has lost vital information required to describe this section, so the programmer must then precede to re-acquire the necessary information.

These two problems of the Breadth-First global control strategy can not be eliminated, however attempts may be made to reduce the problems by providing the programmers with software tools that (1) provide alternative and equally powerful methods of testing the solution as the method of compiling and running; and (2) provide the programmers with the information they need to describe a section of code and in a manner that can readily be comprehended.

7.3.2. Comparing the Performance of Programmers using the Breadth-First Global Control Strategy against those who did not

A third reason why programmers may choose not to use the Breadth-First global control strategy when given a choice is that the solution achieved using the Breadth-First global control strategy may not be as good as the solutions achieved using an alternative control strategy. In this section the performance and behaviour of the programmers using the Breadth-First global control strategy is compared against those of programmers who did not conform to the Breadth-First global control strategy.

7.3.2.1. Completion of Solution

Each participant in the study was given 60 minutes to complete the task presented to them. If a participant had not completed the task in 60 minutes they were asked to stop. Out of the 13 participants who took part in this study only four participants produced a complete solution that fulfilled all the requirements outlined in the task. Two of the four participants who used the Breadth-First global control strategy completed the whole task, while only two of the nine participants who used alternative strategies completed the whole task.

To provide an indication of how much of the solution the participants completed, each participant was scored according to how many of the sub-tasks he or she had
Using the Breadth-First Global Control Strategy

completed. If a participant had made a significant start on a sub-task, but had not completed the sub-task then the participant was given a half mark for that sub-task. As the sub-tasks are not equally difficult, the score of completion is not an ordinal value. The maximum an individual can score is five, a mark for each of the five sub-tasks.

<table>
<thead>
<tr>
<th>Control Strategy Used</th>
<th>Mean Completion Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Breadth-First</td>
<td>3.39</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>4.50</td>
</tr>
</tbody>
</table>

The results in Table 7.2 indicate that the programmers who used the Breadth-First global control strategy on average completed a greater portion of the task than the programmers who diverged from the Breadth-First global control strategy. This is confirmed at the 95% confidence level by an one-tail Independent T test \( (d.f. = 11, t = 1.82) \) performed in Section 1.9. on the individual completion scores that are also report in Section 1.9.

7.3.2.2. Errors in Solution

Although the participants using the Breadth-First global control strategy appear to be completing more of the problem than the participants who do not use that strategy, it is possible that this gain was achieved at a price.

Besides creating a solution for each sub-task it is also important that the solution created is error free. As the programmers using the Breadth-First global control strategy had to parsed more information than the other programmers, it could be hypothesised that this provided a greater opportunity for errors to occur through misunderstanding the information parsed.

Gilmore (1991) identified that errors can full into four categories (surface, control, plan, and interactions). In this study the four categories were slightly altered and two additional categories were introduced. In order to discover an error it is necessary to know what errors can occur. The six following categories were used to guide the discover of the errors.

- Syntax Errors (similar to surface errors) - these errors are superficial that result from the programmer not applying the correct grammatical rules of the programming language to the construction of individual instructions.
- Data-Flow Errors (similar to plan errors) - these errors concern the improper use of data. This includes failing to initialise a variable, incrementing a variable by the wrong amount, incrementing the wrong variable, etc.
- Control-Flow Errors - these errors concern executing segments of the code at inappropriate circumstances. This principally involves an error in the test
conditions that determine whether or not to execute particular segment of codes, or placing the test in the wrong place (e.g. the test may be require to appear at the start of a loop, and not at the end of a loop).

- Inconsistency within a Procedure - an example is a procedure that returns an integer, but where the header of the procedure does not state that it will return an integer.

- Inconsistency between Procedures (similar to interaction errors) - these errors are miscommunications between procedures. For example, the parameters between the calling procedure and the receiving procedure do not match; or a procedure calls another procedure on the presumption that it performs a particular task when in fact it performs a completely different task.

- Specification Errors - i.e. the program does what the programmer intends the program to do, but the actions do not match those mentioned in the specification of the program.

Using these six categories the errors in each of the participant’s solutions were uncovered and categorised.

Table 7.3: Occurrence of Errors for each Control Strategy

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Mean Number of Errors for each Strategy Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Breadth-First</td>
</tr>
<tr>
<td>Syntax</td>
<td>0.50</td>
</tr>
<tr>
<td>Data-Flow</td>
<td>1.00</td>
</tr>
<tr>
<td>Control-Flow</td>
<td>0.25</td>
</tr>
<tr>
<td>Inconsistency within a Procedure</td>
<td>0.25</td>
</tr>
<tr>
<td>Inconsistency between Procedures</td>
<td>1.25</td>
</tr>
<tr>
<td>Specification</td>
<td>0.25</td>
</tr>
<tr>
<td>Overall</td>
<td>3.50</td>
</tr>
</tbody>
</table>

From Table 7.3 it appears that the programmers who do not use the Breadth-First global control strategy are more prone to make mistakes than the programmers who use the Breadth-First global control strategy. Section 1.8. reports the number of errors produced by each participant, and reports the calculation of an one-tail Independent T test. The T test \((d.f. = 11, t = 2.19)\) shows that participants using the Breadth-First control strategy produce significantly (at the 95% confidence level) fewer errors than the other participants.
A possible explanation for this apparent superiority of the Breadth-First global control strategy is that a greater proportion of the Breadth-First global control strategy users had completed the task than users of alternative strategies. Perhaps the participants who had completed the task performed some error checking while those who had not completed the task had not had a chance to check for errors. Indeed the participants who had completed the task made on average 4.25 errors, while the participants who had not completed the task made on average 6.00 errors. A completed solution is likely to have few errors than a near completed solution.

However the difference in the occurrence of errors between the users and the non-users of the Breadth-First global control strategy can not be totally explained by the fact that the users of the Breadth-First global control strategy were more likely to complete the task than non-users. Firstly, the average occurrence of errors for users of the Breadth-First global control strategy is lower than the occurrence of errors for participants who completed the task. Secondly, the users of the Breadth-First global control strategy can not correct mistakes in the program at a higher level than the level they are working on. So when users of the Breadth-First global control strategy has constructed a solution, they can only correct mistakes on the final level.

A possible alternative explanation for the low occurrence of errors for the users of the Breadth-First global control strategy is that the programmers are careful of avoiding errors because they realise errors can not be corrected once they have moved down to the next level of abstraction. Therefore Breadth-First global control strategy users are likely to engage in error detection at each stage the user is contemplating to move down a level. While non-users may only engage in error detection once they have constructed the whole solution, especially when working against the clock. This is borne out by the equally low occurrence of errors (3.50) for users of the Breadth-First global control strategy who have not completed the task as compared to the users of the strategy who had. While the occurrence of errors (6.71) for non-users of the Breadth-First global control strategy who have not completed the task is slightly higher than the occurrence of errors (5.00) for the non-users who had completed the task.

A surprising result is that the occurrence of inconsistency between procedures is approximately the same for both users and non-users of the Breadth-First global control strategy. This is surprising because a major source of interaction between procedures is between the parent procedure and the child procedure, i.e. the error is an inconsistency between two different levels. Participants using the Breadth-First global control strategy when confronted with such as error can only alter the lower level procedure in an attempt to accommodate the higher level procedure. However it is possible that it is the higher level procedure which needs be altered, but the Breadth-First global control strategy prevents such an alternation from being made.
Using the Breadth-First Global Control Strategy

As the inconsistencies between procedures occur just as often in the solution of users of the Breadth-First global control strategy as the solution of non-users, the users of said strategy must be deliberately avoiding the occurrence of this particular error. This can only be achieved by the programmers fully understanding the procedures on the level they are working on which includes understanding the purpose of the procedures which are referred to on the next lower level of abstraction, and making sure that the purposes of these procedures are compatible with the solution of the procedures on the current level of abstraction.

To summarise the users of the Breadth-First global control strategy are likely to engage in error detection before preceding down to next level of abstraction, as only errors on the current level of abstraction can be amended. Particular attention must be paid to whose errors that can span across more than one level abstraction. While the programmers who do not use the Breadth-First global control strategy can be more lax about the occurrence of errors as their errors can be amended once a solution has been created.

It can be argued that it is better to detect errors early rather than later, as certain errors (especially inconsistency between procedures) can be propagated through the solution which makes errors more difficult to remove the longer they remain.

7.3.2.3. Movement between Procedures

Support for the hypothesis that users of the Breadth-First global control strategy are careful in making sure that the solution they produced is correct can be derived from the behaviour of the users when they finished one procedure and started on another procedure.

An indication of how carefully the programmers considered a sub-problem is the time they spent deliberating on the sub-problem before entering the solution into a text editor. When each participant in this study moved from one procedure to another the time was measured between the final keystroke in producing a solution for the previous procedure and the first keystroke in producing a solution for the next procedure. The first keystroke in the production of a procedure was ignored if it was followed by a pause longer than five seconds.

Table 7.4: Time Spent Deliberating

<table>
<thead>
<tr>
<th>Strategy Used</th>
<th>Time (sec.) between spent deliberating at the start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for all procedures</td>
</tr>
<tr>
<td>Not Breadth-First</td>
<td>923.58</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>682.01</td>
</tr>
</tbody>
</table>
Using the Breadth-First Global Control Strategy

From the measurements taken it was found that the users of the Breadth-First global control strategy spent a longer time deliberating at the start of a procedure than the non-users.

Although the users of the Breadth-First global control strategy spent a longer time deliberating at the start of production of each individual procedure than the non-users, the users spent less overall time deliberating than the non-users. Section J.6. reports the time that each participant spent deliberating at the start of a procedure. Section J.6. also reports the calculation of an one-tail Independent T test. The T test \( d.f. = 11, t = 3.18 \) shows that the participants using the Breadth-First control strategy spent significantly (at the 95% confidence level) more time deliberating at the start of a procedure than did the other participants:

The users of the Breadth-First global control strategy made fewer movements between sub-problems than non-users. The principal reason for this is that the Breadth-First global control strategy did not allow the users to return to the solution of previous sub-problem to make alternation, thus reducing the scope of movement and the reason for moving between sub-problems. The movement to previous solutions is the most common type of movement for the non-users of the Breadth-First global control strategy.

### 7.3.2.4. Rate of Construction of Sub-Problems

In addition to the deliberation that programmers engage in at the beginning of a sub-problem, the programmer also has to make further deliberation during the production of the sub-problem's solution. The reason for this that unforeseen difficulties many occur during the production stage which require an alteration in the planned solution.

A comparison of deliberation required during the production stage between the users of a strategy and non-users can be gain by comparing the average time between keystrokes. The faster the keystrokes occur than the less time the programmer spent deliberating on his or her next action.

<table>
<thead>
<tr>
<th>Strategy Used</th>
<th>Average Time (sec.) Between Keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Breadth-First</td>
<td>1.63</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>1.45</td>
</tr>
</tbody>
</table>

The data in Table 7.5 was calculated from the time between all consecutive keystrokes, except where the consecutive keystrokes occurred in the production of different procedures. Section J.7. reports the average time between keystrokes for each participant. In addition section J.7. calculates an one-tail Independent T test.
Using the Breadth-First Global Control Strategy

The T test \((d.f. = 11, t = 0.45)\) indicates that during the solution's production stage of a sub-problem, the users of the Breadth-First global control strategy spent the same amount of time deliberating as the non-users.

7.3.3. Summary

The data collected in this study shows that programmers are reluctant to employ the Breadth-First global control strategy. The only instances where the participants used the Breadth-First global control strategy are when they were given an editor that did not allow them to employ any other global control strategy. The two primary causes of this reluctance are that (1) the Breadth-First global control strategy prevents the users from employing the compile and execute method of testing the solution; (2) and the strategy requires the programmers to sometimes shift attention from one sub-problem to another sub-problem where the information used and gained from the previous sub-problem has no bearing on the preceding sub-problem.

It was also argued that the participants avoided the Breadth-First global control strategy because the solutions produced while using that strategy are not as good as the solutions produced when using an alternative strategy. However the results in this study indicate that the opposite is the case.

The participants forced to use the Breadth-First global control strategy completed more of the task and with fewer errors than the participants who used alternative strategies. A possible reason for this result is that the Breadth-First global control strategy forced its users to deliberate longer at the beginning of a sub-problem in order to avoid creating errors which could not be amended at a later stage. By deliberating longer at the start of the sub-problem, not only did the programmer avoid errors, but it also enabled the programmer to produce the solution quickly without having to stop to make last minute alternation to the planned solution.

As the users of the Breadth-First global control strategy made fewer movements between procedures and produced the procedures quickly after some deliberation, they were able to complete more of the problem task with fewer errors than the non-users of Breadth-First global control strategy.
Chapter 8

General Discussion
Contents of Chapter

In this chapter the questions and answers covered by this thesis are reviewed and critiqued. Section 8.1 outlines the questions asked, and the purpose for asking these questions. Section 8.2 summarises how the questions were answered and what the answers to the questions were. Sections 8.3 and 8.4 discuss the potential implications the data collected has on the programming of psychology literature and on the software industry. Section 8.5 describes the questions posed by the data collected that are related but beyond the bounds of this thesis. This followed by a critique of the method for collecting data in Section 8.6. Finally, Section 8.7 provides a conclusion of research.
8. General Discussion

8.1. Aim of Research

The aim of the research reported within this thesis was to identify aspects of the cognitive processes that underlie the activity of designing programs, with a view to applying the data collected to provide a basis from which to improve the design of software tools. Improved software tools ought to enable programmers to be more productive and to produce higher quality software than current software tools allow them. With this aim in mind, the studies reported within this thesis address (1) in what order software engineers construct solutions, (2) why software engineers construct software in the order that they do, (3) how effective is the Breadth-First control strategy, and (4) how can the use of the Breadth-First control strategy be encouraged.

8.2. Review of Research

8.2.1. Review of Methodology

As stated in the previous section, the aim of the research is to gain information on how programmers typically design software. If the data is affected by observing it, then it is not possible, as wished, generalise the results gathered as being the common strategies of the programmers. This entails that the methodology employed during for the research should be such that it has no impact on modus operandi of the programmers, or that the impact is only on aspects of the programming behaviour that are not relevant to this research, such as the speed of programming.

In order to prevent the methodology of the experiment being a confounding variable influencing the data collected, the methodology was designed to mimic the programming environment in which the participants would commonly engage in programming activities. Attempting to make the experimental setting match the participant's typical programming environment affected the design of the procedure.

As part of the experimental procedure for the first experiment, the participants performed the experiment in their own offices. For the other experiments, the participants performed the experiment in research laboratories at Loughborough and Lancaster University, where they were allowed to adjust the environment so that they felt comfortable. In both cases the aim was to reduce unusual distraction that may alter the programmers' behaviour from their norm. In addition, for all the experiments the participants were allowed to use any equipment that they would normally use, this might include pen and paper, manuals, and software tools such as debuggers. The only restriction was that the participants had to use a particular text editor (Logger-2 or a version of TextRec) and they could not use any other text editor.
General Discussion

As well as the prescribed text editor, the only other factor of the procedure which differ in the experimental environment from the typical programming environment was the requirement for programmers to talk aloud as they attempted to solve the programming task presented to them. Both the prescribed text editor and the requirement to talk aloud were introduced to the procedure in order to collect data. It is argued below that these two alternations to the programming environment did not alter the programming behaviour of the participants from their normal method of programming.

8.2.1.1. Concurrent Verbal Reports

In order to gather information about the cognitive processes of programmers during programming, the participants were asked to think aloud as they tackled a programming task. This request introduced an additional task for programmers to perform and it may influence the programmer's performance and/or behaviour on the primary task of programming. If this is the case then the data collected regarding performance and behaviour of the primary task may not be generalised to programming tasks were concurrent verbal reports were not produced. Ericsson and Simon (1980) investigated the possibility of concurrent verbal reports altering the data collected. They concluded that the influence of the concurrent verbal reports on the primary task depended on how closely the data generated by the primary task matched the data needed to generate verbal reports. The results of their investigation suggest that producing verbal reports while programming may affect the programmers' performance by slowing them down, but it would not affect their behaviour.

An additional consideration for requesting participants to produce concurrent verbal reports is the method by which participants are prompted to produce verbal reports. The danger is that participants produce reports about meta-thoughts (i.e. reporting about thinking) rather than reports about what they are thinking. Therefore, the prompts to speak aloud had to be such that they did not induce the participants to think about what they are thinking. The prompts took the form 'please continue talking', rather than 'what are you thinking'.

8.2.1.2. Text Editors

To collect data on the order in which the programmers constructed their solutions it was necessary to record the interactions between the programmers and the computer. To record all the interactions initiated by either the programmer or the computer, and to record the context in which the interactions occurred it was necessary to develop an environment where such interactions were recorded without interfering with the programmer's use of the computer. To this end TextRec was developed. To the programmers, TextRec appears to be identical to TeachText, a text editor with only basic functions. As TextRec is simple, the programmers can quickly and easily become
acquainted with it. TextRec automatically records the man-machine interactions. The recordings are accurate enough for another application TextView, developed by the author, to read the record of interactions and mimic the interaction between the programmer and TextRec.

8.2.2. Review of Behaviour of Prolog Programmers

The aim of the first experiment was to investigate the global control strategies used by Prolog programmers. The experiment was designed to answer two questions (1) do programmers pursue goals in an orderly fashion that is dependent on the structure of the goal hierarchy (i.e. make use of structured control strategies), and (2) is the Breadth-First global control strategy the predominate strategy used by programmers. If the answer to either of these questions was 'no', then an alternative explanation of programmers' behaviour as observed during the experiment needs to be provided.

The order in which four expert Prolog programmers constructed their solution to a problem was analysed. Each transition was examined to see whether it conformed to a transition prescribed by one of the global control strategies. A transition could conform to more than one structured control strategy. After examining each transition it was possible to calculate the percentage of transitions that conformed to any one global control strategy.

8.2.2.1. Global Control Strategies

The results showed that programmers did not exclusively use one global control strategy but rather they used a combination of two control strategies, the Children-First and the Depth-First global control strategy. This was contrary to the prediction that programmers would use the Breadth-First global control strategy. An explanation of the result is that when programmers select a strategy to use, there are two competing factors. Firstly, programmers wish to use the control strategy that provides them with information on the suitability of the solution to the sub-problem that they are currently working on in comparison to the solutions of sub-problems they have already described. Secondly, programmers wish to use a control strategy which is not so cognitively expensive to follow that they are left with few cognitive resources for other aspect of programming task such as heuristic strategies. Unfortunately for programmers the global control strategy which provides the programmer with the most useful information also requires the most cognitive effort to follow it, while the least demanding global control strategy is also the least effective. It is likely that programmers fail to use the Breadth-First global control strategy because it was too demanding.

Furthermore programmers switched between the Children-First and Depth-First control strategy based on ecological variables. The choice of strategy depended on the
programmer's information needs. When the programmers found themselves in a situation that required them to gain information about the interactions between various components of the solution then they chose to use the Children-First strategy, since its cognitive cost was affordable and it answered many of the programmer's questions. When the programmers were in a situation where information about interaction between various components of the solution is not required, then they will use the Depth-First global control strategy, since it had a low cognitive cost, allowing programmers to apply cognitive resources to other aspect of the programming activity.

8.2.2.2. Transitions Not Conforming to a Global Control Strategy

From the calculation of conforming to either the Depth-First or the Children-First global control strategy it was found that 82% of the transitions conformed to one of these two control strategies. There remained 18% of transitions that did not conform to one of these two control strategies. This begged the question 'what was the cause of these divergent transitions'. Each divergent transition was examined, and from this examination eight causes of divergent activity were discovered these were (1) parent not coded, (2) jumping back to a structured approach after a previous divergent activity, (3) making use of an analogy, (4) debugging, (5) deciding to solve an easier goal to ones prescribed by the global control strategies, (6) adding pre-requisites, (7) adding post-requisites, and (8) working bottom-up.

8.2.3. Review of Behaviour of C Programmers

From the Prolog study it was discovered that programmers use a combination of Children-First and Depth-First global control strategies. As Prolog is an atypical programming language it could be argued that the results from the Prolog study are not generalisable to other programming languages. In response to this potential argument, the Prolog experiment was replicated, but the use of Prolog was replaced with the use of C. Since Prolog and C are very different languages any commonalty of behaviour between Prolog and C programmers can reasonable be assumed to be caused by the programming activity in general, rather than to be caused by the use of a particular programming language.

8.2.3.1. Global Control Strategies

The results from the C study showed that C programmers, like the Prolog programmers, also use a combination of the Children-First and the Depth-First global control strategies. However, the C programmers seem to conform to a global control strategy less often than Prolog programmers. Approximately 5% more transitions were Opportunistic during C programming than during Prolog programming.
8.2.3.2. Frequency of Opportunistic Transitions

From analysis of the Opportunistic transitions it appears that the extra 5% of transitions that are Opportunistic are the result of extra debugging transitions and extra prerequisite transitions. It is possible that the increase in debugging transitions and prerequisite transitions was caused by the nature of the programming language C. The occurrence of debugging transitions is higher for C than Prolog because C is more syntactically complex than Prolog resulting in more errors which need debugging. While, the occurrence of prerequisite transitions is higher for C than Prolog because C possesses variables which need to be declared and initialised, so when any new variable is included in a C program, the prerequisite of declaration and initialisation need to be added.

8.2.3.3. Occurrence of Opportunistic Transitions

In addition to comparing C programming activities to those of Prolog, analyses where performed on when Opportunistic behaviour occurs during programming. The data suggest that the majority (59%) of Opportunistic transitions occur in groups (clusters). The programming activities that cause cluster of transitions to occur are (1) beginning to solve the problem, (2) beginning to solve a sub-problem, (3) changing the solution of a sub-problem, (4) making use of analogies, or (5) debugging the solution of a sub-problem. Beginning to solve the problem, or a sub-problem probably resulted in clusters of Opportunism because the programmers were uncertain of the solution which meant that the programmers had first to explore the problem space. The other activities caused clusters of Opportunism as the programmers' focus of attention is not confined to one locality but to many, in that they have to work on two or more parts of the goal hierarchy that distantly related in terms of control dependency. For example to complete the activity of using analogies the programmer has to continuously move between two parts of the hierarchy, where the two parts are semantically similar but possible distantly control dependency related.

Of the various types of Opportunistic transitions, prerequisite transitions and jump back transitions were more likely to occur outside a cluster, while all the other types of Opportunistic transitions were more likely to occur inside a cluster. In particular debugging transitions were significantly more likely to occur in a cluster than in isolation.

As well as Opportunism occurring in clusters, the occurrence of certain types of Opportunistic transitions was frequently followed by another type of Opportunistic transition. For example 77% of debugging transitions are followed by another debugging transition. While a prerequisite transition is often followed by a jump-back transition (42%) or by another prerequisite transition (38%). The frequent occurrence of particular pairs of Opportunistic transitions results from programmers following
program relationship other than control dependency. For example, a pre-requisite transition followed by another pre-requisite transition occurs when programmers are following data dependency. Consecutive debugging transitions could result from programmers following one of a number of program relationships such as semantic, or data dependency.

8.2.4. Review of Local Control Strategies

The first two studies were concerned with the global control strategies of programmers, which identify a set of potential sub-goals for the programmer to follow. This poses the question of how programmers select a sub-goal to follow from the proposed set of sub-goals. A local control strategy is any method that programmers use to guide their selection of a sub-goal from the set of sub-goals proposed by a global control strategy.

The third study, as described in Chapter 6, attempted to answer the question of what local control strategies do programmers use. From previous claims in the literature, local control strategies based on three factors were considered (1) ease of sub-goals, (2) dependence of sub-goals on other sub-goals, and (3) presence of analogous sub-goals. These three factors provided a possibility of six different local control strategies.

- **Least Dependent First** – the sub-goal selected from the set is the one with least dependence on other proposed sub-goals.

- **Easy First** – the sub-goal selected from the set is the one which is perceived to the easiest to completed.

- **Difficult First** – the sub-goal selected from the set is the one which is perceived to the most difficult to completed.

- **Analogues then Least Dependent First** – the sub-goal selected from the set is the one where an analogy between that sub-goal and a completed sub-goal exists, if there are no perceived analogies then the least dependent sub-goal is selected.

- **Analogues then Easy First** – the sub-goal selected from the set is the one where an analogy between that sub-goal and a completed sub-goal exists, if there are no perceived analogies then the easiest sub-goal is selected.

- **Analogues then Difficult First** – the sub-goal selected from the set is the one where an analogy between that sub-goal and a completed sub-goal exists, if there are no perceived analogies then the most difficult sub-goal is selected.

In this study eight subjects were required to complete four sub-goals. The participants would be conforming to the Children-First, the Breadth-First, and the Depth-First global control strategy no matter what order they chose to complete the four sub-goals. From the previous studies it is known that programmers principally use one of these Top-Down global control strategies. Therefore the order in which they chose to complete the four sub-goals must be guided by a local control strategy.
Each of the 6 proposed local control strategies provide a different prediction of order in which the 4 sub-goals are selected. By comparing the prediction of the proposed local control strategies against the actual behaviour of the participants it is possible to determine which local control strategy programmers do use.

The behaviour of all eight participants matched the prediction of the least-dependent-First local control strategy. The behaviour of the eight participants did not match with any of the other five proposed local control strategies, the only exception is Cl's behaviour which also matched the prediction of the most-difficult-First strategy. The data from the third study overwhelmingly suggests that programmers use the least-dependent-First local control strategy.

8.2.5. Review of Encouraging Breadth-First Global Control Strategy

From the results of the previous chapters, it is argued that programmers select a global control strategy on the basis of two conflicting factors (1) cognitive effort of using the global control strategy, and (2) effectiveness of the global control strategy for producing a well-defined solution. The most computational effective strategy, Breadth-First, also requires the most cognitive effort, perhaps more than programmers can afford to devote to the programming task. The least effective strategy, Depth-First, also requires the least cognitive effort. The Children-First global control strategy provides a middle ground between these two extremes. Programmers wish to employ the least demanding global control strategy, but they also wish to employ the most effective control strategy. As a result of these competing wishes the programmers in most situations select to use the Children-First global control strategy. The difficulty and effectiveness of a strategy does nevertheless depend on the current programming ecology. Consequently, the strategy selected may change through the course of a programming task.

The aim of the study described in Chapter 7 was to reduce the cognitive effort of using the Breadth-First strategy, such that when a programmer selects a global control strategy the Breadth-First strategy becomes the most desirable. It is argued that the Breadth-First strategy is the most demanding because the programmer's shift of focus is greater (distance of transition) for the Breadth-First than for the other control strategies. As a consequence the programmer using the Breadth-First strategy has to parse more information at each shift of focus (transition) than the other programmers using an alternative strategy. The parsing demand of the Breadth-First global control strategy is a phenomenon which cannot be reduced, but it is possible to provide software tools that facilitate programmers to find the information that needs to be parsed, and thus reduce the programmer's cognitive resources devoted to searching for the appropriate information.
General Discussion

Three different editors were designed, one which was a standard text editor, one which supported the use of the Breadth-First control strategy, and one which enforced the Breadth-First control strategy. Each participant was given one of the three text editors to use to complete the programming task. The behaviour of each programmer was measured to see whether they conformed to the use of the Breadth-First control strategy. Although the supportive editor reduced the cognitive effort of using the Breadth-First control strategy, the results showed that its users still avoided using the Breadth-First control strategy as much as the users of the neutral text editor. Naturally, the users of the text editor that enforced the Breadth-First control strategy used the Breadth-First control strategy.

Nevertheless, the Breadth-First control strategy’s advantage of producing better results than the alternative strategies was demonstrated in the experiment described in Chapter 7. The programmers who were forced to use the Breadth-First control strategy produced more complete solutions with fewer errors than the programmers who chose not to use the Breadth-First control strategy.

This suggests that it would be advantageous to develop a software environment that enables programmers to use the Breadth-First control strategy by reducing its cognitive cost. As this would result in programmers being more productive and producing higher quality solutions than they do currently.

8.3. Comparison of Results With The Literature

Within the literature there are a number of studies that observed Opportunistic behaviour that is contrary to the results reported within this thesis. In addition, the literature that provides evidence for a structured approach, argued that programmers use either the Breadth-First or the Depth-First control strategy, this also contradicts the results reported in this thesis. The difference between the results in this thesis and those in the literature may be due to differences of interpretation of the data. The results in this thesis have been interpreted as evidence of programmers using a Children-First control strategy. Since Children-First has not been previously recognised it is possible that when researchers have observed Children-First behaviour that they have interpreted it as being either Breadth-First or as being Opportunistic. Rist (1995) has attempted to solve the dilemma by construct a general theory that explains both Opportunistic and structured design strategies. Rist’s model can also be used to explain Children-First behaviour. Below is a discussion comparing the results of this thesis with those of Opportunistic studies, structured studies, and with Rist’s cue-based search theory.

25 A design strategy explains the order in which goals are recognised and the order in which goals are described. A control strategy also explains the order in which goals are described, but not the order in which they are recognised.
There is also an explanation of the extension made in Chapter 7 to the parsing-gnisrap model.

8.3.1. Comparison with Opportunistic Studies

Both Guindon (1990a) and Visser (1990) reported that experts design software in an Opportunistic fashion rather than systematically exploring the problem space using some predetermined strategy. Guindon measured Opportunistic behaviour as the percentage of transition from one node to another that diverged from a structure Top-Down approach. This measurement was adopted in this thesis, and the average number of observed Opportunistic transitions performed by Prolog and C programmers was 21%. This is far less than Guindon's report of 53% of transition being Opportunistic. However as noted by Ball & Ormerod (1995), Guindon had mistakenly classified Depth-First transitions (which Guindon called drifting) as not being Top-Down. A third of the reported Opportunistic transitions were Depth-First. When Guindon's results are adjusted by re-classifying the Depth-First transitions as being Top-Down, Guindon's data indicates that 35% of the transitions was Opportunistic. This lessens the disparity between Guindon's results and those reported in this thesis. In addition, Guindon's study investigated the early stages of the design process whereas the studies within this thesis investigated all stages of design. Given that the results in Chapter 5 show that programmers behave Opportunistically when they start to solve a new problem before moving to structured strategy, it is not surprising that Guindon's study showed a higher percentage of Opportunistic behaviour than the studies within this thesis.

Visser (1990) considered any activity transfer to be Opportunistic if it diverged from the structured control strategy which the designer stated he would follow. The designer Visser investigated stated that he would use the Depth-First control strategy. Therefore if the designer employed an alternative Top-Down approach such as Breadth-First or Children-First, that approach was considered to be Opportunistic. From Visser's reported results it appears that 24% of activity transfers diverged from the Depth-First approach and that they were labelled as being Opportunistic activity transfers. From the Prolog and C studies an average of 33% of transitions diverged from the Depth-First control strategy. If the alternative structured control strategies were considered, then Visser's data would probably show even fewer Opportunistic activity transfers. It should be noted that transitions and activity transfers are not directly comparable, but it does seem that both studies show a preponderance of Top-Down behaviour.

8.3.2. Comparison with Structured Studies

The studies which have reported experts using a Breadth-First approach to decompose a programming problem (Jeffries et al., 1981; Adelson & Soloway, 1985; Rist, 1990)
based their conclusions on qualitative data. This thesis supplements their qualitative data with further qualitative data and quantitative data. The additional data suggests that programmers do not use the Breadth-First approach but rather they use the Children-First approach. A likely explanation in the difference of results is the similarity of Breadth-First behaviour and Children-First behaviour for small problems. Breadth-First and Children-First are identical for problems that consist of only three levels of decomposition. A problem decomposed into four levels by programmers using the Children-First approach, could also be explained as the programmer using the Breadth-First strategy with some Depth-First deviations.

It is only when programmers are given problems that can not be decomposed into less than five levels of abstraction that differences between the two strategies become clear. The problems employed in this thesis were complicated enough to differentiate between the different strategies, whereas the problems used by previous researchers who witnessed a Top-Down approach were less complex. Their data were such that the overt behavioural differences between the two control strategies would not exist. Consequentially, researchers who interpreted their data as exhibiting Breadth-First behaviour were probably witnessing programmers using the Children-First approach.

8.3.3. Comparison with Rist’s Cue-Based Search Theory

Rist (1995), in agreement with this thesis, viewed programs as a single entity made up of inter-connecting nodes. While this thesis concentrated on control flow connections between nodes, Rist paid equal attention to data flow connections. By only concentrating on transitions following specified control flow links, it is possible to account for about 80% of transitions between nodes. A model, such as Rist’s, that includes the possibility of transitions that follow data flow links as well as transitions that follow control flow links could account for the 80% of transitions that follow control flow links and account for some of the transitions that were labelled divergent within this thesis. For example, Rist’s model allows for the pre-requisite and the post-requisite transitions. Rist’s model can account for a greater proportion of transitions than the thesis’ proposed model of programmers using a combination of the Children-First and the Depth-First control strategies to traverse control flow links.

Although the explanatory power of Rist’s model is greater than the one presented within this thesis, the predictive power of Rist’s model is weaker. Rist’s model explains that programmers follow a design strategy from a start cue generated from a node. Rist does not specify what design strategies programmers use, but Rist’s model does limit the possibilities, this means that Rist’s model can not be used to predict behaviour without first providing his model with a predetermined design strategy.

Rist’s model allows for a design strategy that would result in behaviour that is similar to a mixture of the Children-First and Depth-First control strategies. Such a
design strategy would specify that programmers work Top-Down, and forward through control-flow links, also that programmers scan backward through their internal and external memories looking for cues. The contents of the memory would determine whether the programmer behaves in a fashion that can be characterised as Depth-First or in a fashion that could be characterised as Children-First. If programmers use a design strategy where they scanned forward through their memories then they would behave in a Breadth-First fashion.

Rist's model can provide an explanation of programming behaviour reported within this thesis. The results described can be used to determine a design strategy to provide Rist's model with the predictive power that it needs to be of use.

8.3.4. An Extension to the Parsing-Gnisrap Model
Green et al (1987) explained that programming behaviour (the generation of code, and the parsing of code) is influenced by the programming language, and in particular the notation of the language. A concern of the parsing-gnisrap model is how easy it is to access information contained within the code. Accessibility of information is influenced by the 'access window'. Green et al (1987) noted that the size of the access window is important for parsing. In Chapter 7, it was noted that the content of the access window is also important. A small window containing needed information is more helpful than a large window containing irrelevant information. Not only is irrelevant information uninformative, but it can also hinder the programmers as they have to waste resources evaluating the relevance of the information. Ideally, an access window should contain as much needed information as possible, and as little irrelevant information as possible.

8.4. Implications of Results For Industry
The ultimate purpose of this research is to improve the working practice of the software engineers, so that they will be more productive and produce better quality solutions than they currently do. The results of the studies described in this thesis provide some tentative indications of how this may be achieved. After results from future investigations have been analysed, these suggestions will become less ethereal.

8.4.1. Training of Programmers
From the results in Chapter 7, it is clear that programmers do not always select the global control strategy that is most effective. This indicates that the programmers lacked the necessary skills to select the best available control strategy, and/or the skills to apply the best global control strategy.

This lack of skills can partly be elevated by providing the programmers with tools that guide and enforce the best control strategies, however the programmer still may not know how to gain most from using the prescribed control strategy. Furthermore, the best control strategy for programmers is dependent on their abilities, such as the
management of their cognitive resources. Thus, software tools can complement programmers' skills, but they can not with current technology supplant programmers' skills. To gain full advantage of global control strategies programmers need to be trained in the various global control strategies, and be taught the pros and cons of each global control strategy so that they can gauge the most appropriate global control strategy to use in any given situation.

8.4.2. Design of Software Tools

A major problem facing the software industry is that little attention is paid to the needs of the users when designing software. This remains true even when the industry designs software to be used by programmers. Often part of the problem is that the needs of the users are not known, and the designers do not have the skills nor the time to determine what the needs of the users are. Even when the designer shares the same needs as the users, the designer's introspection is frequently faulty. Experimental studies on programming behaviour, such as the ones presented in this thesis, can provide the designer with information about the needs of the programmers.

In order for a software tool to be advantageous to the programmer, it must readily provide the programmer with information that can be put to beneficial use. Typically, this requirement of software tools is seen to be achieved by identifying what information the programmer needs in any given situation, and by identifying in what form the information should be presented. While these are important criteria for determine the effectiveness of a software tool, there is another criterion that is no less important. Just as software tools should provide advantageous information it should also guide the programmers to the situation were the information that it can provide will be most beneficial.

The results of the studies described in this thesis suggests that a useful software tool is one which enforces a particular global control strategy, and one which provides context (relevant information) for the sub-problem that the programmer is working on. A software tool that enforces the Breadth-First global control strategy would place the programmer in a situation (a location within the goal hierarchy) that is advantageous to the solution being produced, as it enables the programmer to capitalise on the useful information concern interaction between modules. The result of such a software tool would improve the productivity of programmers, and it would improve the quality of the software produced.

The thesis also argues that programmers require information that provides programmers with a description of context in which they are working. The supposition is that the sections of the solution that are antecedents to the current sub-problem provide the necessary context for programmers to produce a well-designed solution to the sub-problem. The antecedents identify the inputs to and outputs from the sub-
problem, and possibly some explanation of the process by which the outputs can be derived from the inputs. The other parts of the solution which are not antecedents to the sub-problem do not contain any information that is relevant to the function of the sub-problem, and therefore only contain information that is extraneous to the sub-problem.

To summarise a software tool needs to provide the programmer with the appropriate information, and to place the programmer in the appropriate situation so that the programmer can fully capitalise on the information to produce a well-designed solution.

8.5. Issues for Further Investigations
The research described in this thesis concentrated on the influence of languages and editors on the control strategies used by programmers. There are many more questions concerning the use of control strategies within programming and within other domains. These questions concern (1) how the choice of control strategies is influenced by other factors such as the expertise of programmers, (2) what is the most effective control strategy in any given situation, and (3) how to apply successfully knowledge gained about control strategies to improve productivity of programmers and to improve the quality of their work. In addition, there is the question of whether the Children-First approach to Top-Down design occurs in other domains.

8.5.1. Factors Influencing Choice of Control Strategy
8.5.1.1. Object-Oriented Languages
The studies undertaken were concerned with languages that typical result in program solutions having a hierarchical structure (or very close to having such a structure). The majority of commonly used languages would show similar characteristics, however there is a paradigm of languages which do not result in hierarchical solutions. Object-oriented languages consist of objects that communicate with each other, the structure of communication is not based on a hierarchy but rather it is based on a directed graph (also known as a net) which has many loops. (A hierarchy is mathematically a directed graph, were every node, except the top node, is at the end of one and only one arc.)

![Figure 8.1: A Tree and a Net](image)

Top-down structured control strategies specify how to traverse a hierarchy, and therefore without some alternations they cannot specify how a directed graph is to be traversed so that all the nodes are visited/recognised. The problem is mathematically
identical to way-finding in a maze with loops, i.e. how to avoid wandering around in circles. A possibility is for the global control strategy to include some mechanism for converting a net structure into a tree structure where all nodes are visited (a tree derived from a net is known as a spanning tree).

By investigating programmers using object-oriented languages, it is possible to determine whether programmers solve the problem of traversing a net by converting it into a spanning tree structure and using a Top-Down control strategy, or whether they solve the problem by using some alternative, as yet unknown, strategy.

8.5.1.2. Cognitive Resources

An argument employed in this thesis is that the programmer’s choice of control strategy is dependent on the availability of cognitive resources (e.g. vacant spaces within working memory). The argument is that programmers want to choose the strategy which produces the best results. Unfortunately the best strategy also happens to be the most costly in terms of cognitive resources, and the programmers’ choice of strategy will be limited to how much cognitive resources they can devote to the control strategy. This needs to be determine empirical whether it is possible to affect the control strategy the programmers select by manipulating the supply of cognitive resources and the demand for cognitive resources. The hypothesis is that programmers will select Breadth-First, then Children-First and then Depth-First as the availability of cognitive resources diminishes.

8.5.1.3. Expertise of Programmers

Another interesting question concerning the choice of control strategies is what influence does the training and experiences of programmers have on their choice. Expertise of programmers may influence choice in several ways such as their ability to manage cognitive resources and information needs.

8.5.2. Most Effective Control Strategy

In order to apply effectively the research on control strategies to improving the work of software engineers, it is first necessary to determine the usefulness of each control strategy in any given situation. The study in Chapter 7 has made an initial start to answering this question by showing that programmers seem to perform better when
they were forced to use the Breadth-First global control strategy than when they chose to use an alternative control strategy. However the study in Chapter 7 only covered one task, if the participants had to tackle a different task it is possible that the Breadth-First strategy would not be the most effective. Furthermore due to time constraints and the lack of resources, the study in Chapter 7 only focused on the use of the Breadth-First global control strategy while ignoring the use of local control strategies or the enforced use of alternative global control strategies. Evaluation of local and global control strategies has only just been started and it is an area of study that could require many more experiments to answer the questions posed by this topic.

8.5.3. Improving Construction of Software

After discovering the factors that influence the choice of control strategies and the effectiveness of each control strategy for any given situation, it is possible to apply the information gained to improving the work of software engineers. However the theories that explain the results of the experiments can never be proven to be true (theories are either false or not yet proven to be false), consequently the practical application of theories is not guaranteed to work. Therefore, testing the application of theories in the real-world is important for two reasons (1) to test whether the theory is false, and (2) to prevent the adoption of new working practices that may be less beneficial than the original methods.

8.6. Critique of Research Methodology

Below is a discussion of (1) problems with the methodology employed within the described studies, (2) the implications of these problems, and (3) how the problems could be resolved in future studies.

8.6.1. Weaknesses of Research Methodology

There were a number of problems with the methodology employed in the described studies. The weaknesses of methodology are that:

1. only one task was used in each studies,
2. the problems used were not large scale,
3. the experiments contained few participants, and
4. the motivation of the participants may be different from the motivation of programmers in the work environment.

In addition to these generic weaknesses, the methodology of the last two experiments contained their own weaknesses. The methodology of the two protocol studies only contains generic weaknesses, as they were purely observational studies where independent variables were not manipulated to examine their effect on dependent
variables. The protocol studies had limited methodology, thus little scope for flaws within the methodology.

The study on local control strategies had a few weaknesses concerning the ability to accurately recognise which local control strategy the participants were using. In particular there was:

5. a lack of distinction of prioritising goals according presentation order and of execution order, and
6. the ease of the goals may have been too similar for participants to be able to prioritise goals according to ease/difficulty of completing the goals.

The study on encouraging the use of the breadth-first control strategy also had a weakness:

7. the environment that enforced the breadth-first control strategy was more viscous (Green, 1989) than the other environments.

These weaknesses had implications on the interpretation of the results, which affects the generalisability on the results.

8.6.2. Implications of Weaknesses for Results

8.6.2.1. Only one task was used in each studies

For both the Prolog and C studies a variation on the signal problem was given to all participants to solve, and no other programming problem was used. In Chapter 4 it was demonstrated that the complexity of sub-problems affects the choice of global control strategy. If an individual sub-problem affects the choice of strategy it is quite likely that the overall pattern of behaviour of the programmer may be affected by the complexity of the overall problem. For example, if a programmer was presented with a more demanding programming problem then the proportion of transitions conforming to the Children-First control strategy may increase while the proportion of transitions conforming to the Depth-First control strategy may decrease; alternatively more Opportunistic transitions may occur. So while the Prolog and C studies demonstrated that programmers do use structured control strategies, the studies do not demonstrate that programmers always predominately use structured control strategies. In short the results from these studies may not be generalised to programmers solving different programming problems.

8.6.2.2. The problems used were not large scale

A related problem to the use of only one problem for the Prolog and C studies, is that the problem used could be solved within two hours. The problem presented to the participants in the other studies described within this thesis, only required one hour in which the problems could be solved. In the work place programmers rarely tackle a
problem which requires only two hours to be solved. Typically programmers work on problems that require days, months or even years to be solved, also they rarely work in isolation but in teams. Large scale projects and working within teams may require the programmers to employ different control strategies to the ones programmers use to solve two hour problems. However the results of the studies may be generalised to the small sub-problems which programmers have to tackle in the work place, although this is dependent on whether programmers solve the sub-problems in isolation from rest of the problem.

8.6.2.3. Few participants
The third problem with the methodology employed is that the experiments contained few participants. This means that the results of the experiments could be strongly influenced by individuals. The results may be more indicative of the behaviour of a few unusual programmers rather than representative of the behaviour of programmers in general. However, given that there were few observed individual differences between the programmers who did participate and the probability that all of the participants exhibit the same eccentricity is remote, it is likely that the participants exhibit patterns of behaviour that are typical of programmers using Prolog, C, or similar programming languages.

8.6.2.4. Motivation
Nevertheless there is one factor that may have differentiated the participants from programmers in the work place and that is their motivation to produce well-designed solutions. Within the work place there are consequences for designing good solutions and for designing poor solutions, such that programmers are typically motivated to produce good solutions. Whereas in the experimental setting participants are not rewarded for producing well-designed solutions, although they may have other motives for producing good solutions, such as pride and a wish to please the experimenter. In any case the level of motivation of participants may be different to that displayed in the work place, and this may affect the control strategies adopted by the programmers. Therefore, the choice of control strategies in the experimental setting may be different to the choice of control strategies in the work place.

8.6.2.5. Presentation and Execution Order Being Identical
The results from the study on local control strategies suggest that software engineers pursue goals according to the order in which they are presented, i.e. when a global control strategy limits the designers choice to a few goals, the designer selects to pursue the goal that is presented first within the program text. Unfortunately, the order in which the goals are presented within the program text is also typically the order in which the goals are executed. This was the case with the task presented to the
participants in the local control strategies study. This meant that the order in which the
participants pursued the goal, agreed with the order in which the goals are executed as
well as agreeing with the order in which the goals are presented.

As a consequence it is not possible to determine whether the participants used a
local control strategy based on the order of presentation, or one based on the order of
execution. Indeed it is possible as these two orders are typical identical that
programmers may follow one as a guide to following the other, e.g. programmers may
select goals according to the order of presentation (as it the easiest of the two to follow),
because they wish to pursue goals according to the order of execution.

8.6.2.6. Similarity of Ease

The second weakness of the local control strategies study is that ease of completing
each of the goals may have been too similar. Given that the task given to the
participants only last for only one hour, the difference of ease between the goals could
not be as great as would normally occur within a design task. The greater the similarity
of ease between the goals, lessens the possible advantages of pursue the goals
according to their ease/difficulty. Within a typical design task, participants may use
ease of completion as a criterion for selecting a goal to pursue. However within the
local control strategies study, the advantages of pursue goals according to ease of
completion may have diminished to such an extent that it is no longer a useful strategy,
thus the participants choice to use a different local control strategy.

8.6.2.8. Viscosity of Environment

The results from the breadth-first control strategy study showed that participants using
the breadth-first control strategy performed better than participants using alternative
control strategies. However, the only participants who used the breadth-first control
strategy were those who were forced to do so by the editor. It is possible that some
aspect of the editor that enforced the breadth-first control strategy was responsible for
the resultant performance of participants other than forcing them to use the breadth-first
control strategy. A side effect of the editor forcing participants to use the breadth-first
control strategy is that the editor becomes viscous, i.e. it becomes difficult to make
alterations to the code that has already been produced. The editor that forces use of the
breadth-first control strategy is more viscous than the other editors. As it is difficult to
make changes to the code in the breadth-first editor, its users may have been more
cautions to avoid mistakes than users of the other two editors.

It is possible that the difference of viscosity between the editors caused the
difference of performance between participants. Therefore, it is not possible with
absolute certainty to ascribe the difference of performance to the control strategy used.
8.6.3. How to Improve Research Methodology

8.6.3.1. How to Improve Generic Weaknesses

To overcome these four generic weaknesses, the methodology could be improved in two respects, these are (1) observing behaviour in the work place, and (2) observing the behaviour of many more programmers tackling many problems.

The potential problems of motivation and of only investigating simple problems are resolved by observing programmers in the work place performing their typical programming duties. While, the data collected from such case studies may be strongly influenced by the idiosyncratic elements of the work place (such as office politics), the data collected will be less influenced by the idiosyncratic elements of studies carried out in laboratories (such as the possible lack of motivation, and the simplicity of problems). However, by performing a number of case studies within different work places, the idiosyncratic elements of each case study can be counteracted.

The other weakness of the methodology, few participants and only one problem, potentially makes the data collected adversely influenced by confounding factors that are determined by the participants and/or the problem used. The probability of confounding factors unduly influencing the results can be lessened by increasing the number of participants observed and increasing the range of problems observed.

8.6.3.2. How to Improve Weaknesses of the Local Control Strategies Study

The weaknesses of the local control strategy study concerned (1) the lack of difference between presentation order and execution order of goals, and (2) the lack of significant difference of the goals’ ease/difficulty.

Since it is typical for the presentation order of goals to reflect the order of execution of goals, any task given to participants that divorced the two orders would be atypical. Any experiment which used such an atypical task would lack some ecological validity. However the experiment would be useful in providing circumstantial evidence of whether software engineers follow goals according to their order of presentation or their order of execution.

The second weakness, the possible lack of significant difference of the goals’ ease, is less difficult to resolve. The framework of the study could be replicated where different goals will have to be used. For the new goals to be able to have a greater scope of difference it would be necessary, (1) to increase the length of the experiment, and/or (2) to reduce the number of goals from four to two.

8.6.3.3. How to Improve Weaknesses of the Breadth-First Study

The weakness of the breadth-first study is that the viscosity of the three editors differ and as the only participants to use the breadth-first control strategy used the editor that
enforced said strategy, it is possible that the good performance of the participants who used the breadth-first control strategy was due to the viscosity of the editor they used and not due to the control strategy they used. In order to resolve the cause of better performance it is necessary to repeat the breadth-first experiment, but ensuring that the viscosity of all three editors is identical. Since high viscosity is a side effect of forcing participants to use the breadth-first control strategy, it is not possible to reduce the viscosity of the editor that enforced the breadth-first control strategy. Therefore, the viscosity of the other two editors needs to be increased by making it equally impossible to change code that has already been produced.

If from the following experiment it is observed that participants who use the breadth-first control strategy still perform better than the other participants then it is possible to conclude that the control strategy used is the cause of the difference since viscosity has been controlled.

8.7. Conclusion

The thesis has reported four studies investigating the role that local and global control strategies play in the design of software. This included how the programming language and the editor affects the control strategies programmers choose to follow, how to encourage programmers to use the Breadth-First control, and whether use of the Breadth-First control strategy improves the quality of software and the productivity of constructing software.

The results showed that programmers do not readily employ the Breadth-First global control, despite it being the commonly prescribed strategy, but rather they use a combination of the Children-First and the Depth-First control strategy. The choice of global control strategy is dependent on the complexity of the sub-problem, and is also likely to be dependent on the limited cognitive resources of the programmer. Programmers occasionally diverged from a Top-Down control strategy in order to follow up on opportunities. These opportunities normally occur in clusters in particular situations, such as when the programmer is in the initial stages of tackling a problem. The pattern of programming behaviour was broadly similar across programming languages.

The investigation into local control strategies showed that programmers describe sub-goals in the order that they appear in the program text, possible because it insures that programmers complete sub-goals in the order of least dependence first. It is possible that there are alternative local control strategies that are better than this one, but that programmers need to spend their cognitive resources on other aspects of programming and avoid alternative local control strategies that are expensive.

The final study investigated the editor's effect on the choice of global control strategy and the effectiveness of the Breadth-First control strategy in comparison the
alternative global control strategies that programmers choose to use. The results showed that reducing the cognitive cost which the Breadth-First control strategy places on programmers is not enough to encourage them to use the Breadth-First control strategy. Addition facilities are need to make programmers willing to adopt the Breadth-First control strategy. Nevertheless when programmers were forced to use the Breadth-First control strategy they performed better than their peers who used alternative control strategies.

Further work is needed to determine which local control strategy is the most effective for producing good quality work, and how to reduce the cognitive cost of the Breadth-First control strategy so that programmers will be encouraged to use it. Although forcing programmers to use the Breadth-First control strategy improved their performance, it also prevents them from capitalising on opportunities as they present themselves and thus improving their performance even further. An ideal programming environment is one that does not force but encourages and supports the use of the optimal local and global control strategy.
Appendices
### Table A.1: Breadth-First Global Control Strategy Decomposition

<table>
<thead>
<tr>
<th>Step</th>
<th>Goal Being Considered</th>
<th>Recognised Goals</th>
<th>Described Goals</th>
<th>Completed Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>cb</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>dc</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>edc</td>
<td>ba</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>fed</td>
<td>ba</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>gfed</td>
<td>cba</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>d</td>
<td>hgfe</td>
<td>cba</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>d</td>
<td>ihgf</td>
<td>dcba</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>e</td>
<td>jihgf</td>
<td>dcba</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>kjihgf</td>
<td>edcba</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>f</td>
<td>lkjihg</td>
<td>edcba</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>f</td>
<td>mlkjihg</td>
<td>fedcba</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>g</td>
<td>nmlkjih</td>
<td>fedcba</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>g</td>
<td>onmlkjih</td>
<td>gfeedcba</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>h</td>
<td>onmlkjji</td>
<td>gfeedcba</td>
<td>h</td>
</tr>
<tr>
<td>17</td>
<td>i</td>
<td>onmlkj</td>
<td>gfeedba</td>
<td>dih</td>
</tr>
<tr>
<td>18</td>
<td>j</td>
<td>onmlk</td>
<td>gfeedba</td>
<td>jdih</td>
</tr>
<tr>
<td>19</td>
<td>k</td>
<td>onml</td>
<td>gfeca</td>
<td>bekjdih</td>
</tr>
<tr>
<td>20</td>
<td>l</td>
<td>onm</td>
<td>gfca</td>
<td>lbekjdih</td>
</tr>
<tr>
<td>21</td>
<td>m</td>
<td>on</td>
<td>gca</td>
<td>fmlbekjdih</td>
</tr>
<tr>
<td>22</td>
<td>n</td>
<td>o</td>
<td>gca</td>
<td>nfmibekjdih</td>
</tr>
<tr>
<td>23</td>
<td>o</td>
<td></td>
<td>aegonfmlbekjdih</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Goal Being Considered</td>
<td>Recognised Goals</td>
<td>Described Goals</td>
<td>Completed Goals</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>abd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>abdh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>h</td>
<td>abd</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>abdi</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i</td>
<td>ab</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>abe</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>abej</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>j</td>
<td>abe</td>
<td>jdh</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>abek</td>
<td>jdh</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>k</td>
<td>a</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>ac</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>c</td>
<td>acf</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>f</td>
<td>acfl</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>l</td>
<td>acf</td>
<td>labekjdh</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>f</td>
<td>acfm</td>
<td>labekjdh</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>m</td>
<td>ac</td>
<td>fmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>c</td>
<td>acg</td>
<td>fmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>g</td>
<td>acgn</td>
<td>fmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>n</td>
<td>acg</td>
<td>nfmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>g</td>
<td>acgo</td>
<td>nfmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>o</td>
<td>acgonfmlbekjdh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Goal Being Considered</td>
<td>Recognised Goals</td>
<td>Described Goals</td>
<td>Completed Goals</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>abd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>abdh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>h</td>
<td>abd</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>abdi</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i</td>
<td>ab</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>abe</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>abej</td>
<td>dih</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>j</td>
<td>abe</td>
<td>jdh</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>abek</td>
<td>jdh</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>k</td>
<td>a</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>a</td>
<td>ac</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>c</td>
<td>acf</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>f</td>
<td>acfl</td>
<td>bekjdh</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>l</td>
<td>acf</td>
<td>lbekjdh</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>f</td>
<td>acfrm</td>
<td>lbekjdh</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>m</td>
<td>ac</td>
<td>lbekjdh</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>c</td>
<td>acg</td>
<td>lbekjdh</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>g</td>
<td>acgn</td>
<td>lbekjdh</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>n</td>
<td>acg</td>
<td>nfmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>g</td>
<td>acgo</td>
<td>nfmlbekjdh</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>o</td>
<td></td>
<td>acgonnfmlbekjdh</td>
<td></td>
</tr>
</tbody>
</table>
### Tables for Protocol Analysis

#### B.1. Generalised Actions

**Table B.1: Protocol Analysis Codes**

<table>
<thead>
<tr>
<th>Purpose of Action</th>
<th>Additional Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A Scheduling &amp; Selecting Problem(s)</strong></td>
<td></td>
</tr>
<tr>
<td>A1 Assessing Ease of Problem</td>
<td></td>
</tr>
<tr>
<td>A2 Explicitly Scheduling Problem(s)</td>
<td></td>
</tr>
<tr>
<td>A3 Explicitly Selecting Problem</td>
<td></td>
</tr>
<tr>
<td>A4 Implicitly Selecting Problem</td>
<td></td>
</tr>
<tr>
<td><strong>B Understanding Problem(s)</strong></td>
<td>B1 Understanding Functional Requirement(s)</td>
</tr>
<tr>
<td>B2 Understanding Input</td>
<td></td>
</tr>
<tr>
<td>B3 Understanding Output</td>
<td></td>
</tr>
<tr>
<td>B4 Inferring New Requirement/Constraint</td>
<td></td>
</tr>
<tr>
<td>B5 Simulating Problem Scenario</td>
<td></td>
</tr>
<tr>
<td><strong>C Generating Solution Concept(s) for Problem</strong></td>
<td>Comparing Solution Concept(s) Against Requirements or Alternative Concept(s)</td>
</tr>
<tr>
<td><strong>D Evaluating Solution Concept(s) for Problem</strong></td>
<td>D1 Simulating Functionality of Solution Concept(s)</td>
</tr>
<tr>
<td>D2 Applying Detailed Knowledge To Evaluate Higher-Level Solution Concept(s)</td>
<td></td>
</tr>
<tr>
<td>D3a Applying Detailed Knowledge To Evaluate Lower-Level Solution Concept(s)</td>
<td></td>
</tr>
<tr>
<td>D4 Compiling/Querying Code</td>
<td></td>
</tr>
<tr>
<td>D5 Tracing</td>
<td></td>
</tr>
<tr>
<td><strong>E Monitoring Progress</strong></td>
<td>E1 Evaluating Progress</td>
</tr>
<tr>
<td>E2 Summarising Progress</td>
<td></td>
</tr>
<tr>
<td><strong>F Externalising Solution Concept(s)</strong></td>
<td>F1 Writing Code</td>
</tr>
<tr>
<td>F2 Writing Notes</td>
<td></td>
</tr>
<tr>
<td><strong>G Searching for Information</strong></td>
<td>G1 Reading Problem Specification</td>
</tr>
<tr>
<td>G1a Reading Manual</td>
<td></td>
</tr>
<tr>
<td>G1b Reading Output</td>
<td></td>
</tr>
<tr>
<td>G1c Reading Code</td>
<td></td>
</tr>
<tr>
<td>G2 Questioning Investigator</td>
<td></td>
</tr>
<tr>
<td>G3 Questioning Self</td>
<td></td>
</tr>
<tr>
<td>G4 Questioning Environment</td>
<td></td>
</tr>
<tr>
<td><strong>H Meta-Comment</strong></td>
<td></td>
</tr>
<tr>
<td><strong>I Others</strong></td>
<td></td>
</tr>
</tbody>
</table>
B.2. Structure of TextRec Log Files

The structure of the log file is outlined below in BNF (Backus-Naur Form).

```
<log file> ::= <record>*

<record> ::= <time> <type> <source> <params>

<time> ::= <LONGINT> (number of ticks since start-up)

?type> ::= <STRING> (see table below for list of possible values of <type>)

<source> ::= Key | Mouse | Menu | Application (source of action)

<params> ::= see Section 3.A.2.1.

<fname> ::= <STRING> (file name)

<wname> ::= <STRING> (window name)

<old_fname> ::= <STRING> (old file name)

<new_fname> ::= <STRING> (new file name)

<insertion point> ::= <POINT> (position of insertion point)

<POINT> ::= <INT> <INT>

<clipboard> ::= <TEXT> (the contents of the clipboard)

<text> ::= <TEXT> (the contents of the window)

<charcode> ::= <INT> (ASCII code of character)
```

* - means object is repeated for unspecified number of times (including zero times).
Λ - a null value.
## B.2.1. Parameters Associated with Each Type of Interaction

**Table B.2: List of Interactions Recorded By TextRec**

<table>
<thead>
<tr>
<th>&lt;type&gt;</th>
<th>&lt;params&gt;</th>
<th>General Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>λ</td>
<td>Select all windows contents</td>
</tr>
<tr>
<td>Beep</td>
<td>λ</td>
<td>Making a sound</td>
</tr>
<tr>
<td>Clear</td>
<td>λ</td>
<td>Clear selected text</td>
</tr>
<tr>
<td>Clipboard</td>
<td>λ</td>
<td>Show Clipboard</td>
</tr>
<tr>
<td>Close</td>
<td>&lt;wname&gt;</td>
<td>Closing a window</td>
</tr>
<tr>
<td>Copy</td>
<td>λ</td>
<td>Copy selected text</td>
</tr>
<tr>
<td>Cut</td>
<td>λ</td>
<td>Cut selected text</td>
</tr>
<tr>
<td>Insert</td>
<td>&lt;insertion point&gt;</td>
<td>Moving to new insertion point</td>
</tr>
<tr>
<td></td>
<td>&lt;charcode&gt;+</td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td>&lt;text&gt;</td>
<td>Editing new procedure, changing text displayed and insertion point</td>
</tr>
<tr>
<td></td>
<td>&lt;insertion point&gt;</td>
<td></td>
</tr>
<tr>
<td>Move-fail</td>
<td>λ</td>
<td>Cancelling move</td>
</tr>
<tr>
<td>New</td>
<td>λ</td>
<td>Opening a new window</td>
</tr>
<tr>
<td>Open</td>
<td>&lt;fname&gt;</td>
<td>Opening a text file</td>
</tr>
<tr>
<td>Open-fail</td>
<td>λ</td>
<td>Cancelling opening a text file</td>
</tr>
<tr>
<td>PageSetup</td>
<td>λ</td>
<td>Configuring Printing</td>
</tr>
<tr>
<td>Paste</td>
<td>λ</td>
<td>Pasting clipboard at insertion point</td>
</tr>
<tr>
<td>Print</td>
<td>&lt;wname&gt;</td>
<td>Print contents of window</td>
</tr>
<tr>
<td>Quit</td>
<td>λ</td>
<td>Quit from TextRec</td>
</tr>
<tr>
<td>Record</td>
<td>&lt;fname&gt;</td>
<td>Start of recording interaction, storing initial state</td>
</tr>
<tr>
<td></td>
<td>&lt;wname&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;insertion point&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;clipboard&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;text&gt;</td>
<td></td>
</tr>
<tr>
<td>Resume</td>
<td>&lt;clipboard&gt;</td>
<td>Reactivating TextRec</td>
</tr>
<tr>
<td>Save</td>
<td>&lt;old_fname&gt;</td>
<td>Saving contents of window</td>
</tr>
<tr>
<td></td>
<td>&lt;new_fname&gt;</td>
<td></td>
</tr>
<tr>
<td>Save-fail</td>
<td>λ</td>
<td>Failing to save contents</td>
</tr>
<tr>
<td>SaveAs</td>
<td>&lt;old_fname&gt;</td>
<td>Saving contents of window</td>
</tr>
<tr>
<td></td>
<td>&lt;new_fname&gt;</td>
<td></td>
</tr>
<tr>
<td>SaveAs-fail</td>
<td>λ</td>
<td>Failing to save contents</td>
</tr>
<tr>
<td>SelectWin</td>
<td>&lt;wname&gt;</td>
<td>Change selected window</td>
</tr>
<tr>
<td>Show</td>
<td>&lt;text&gt;</td>
<td>TextRec displaying information</td>
</tr>
<tr>
<td>Stop</td>
<td>λ</td>
<td>Stop recording</td>
</tr>
<tr>
<td>Stop-fail</td>
<td>λ</td>
<td>Cancelling stop of recording</td>
</tr>
<tr>
<td>Suspend</td>
<td>λ</td>
<td>Deactivating TextRec</td>
</tr>
<tr>
<td>Type</td>
<td>&lt;charcode&gt;</td>
<td>Entering a character</td>
</tr>
<tr>
<td>Type-fail</td>
<td>&lt;charcode&gt;</td>
<td>Entering a character which can not be accepted</td>
</tr>
<tr>
<td>Undo</td>
<td>λ</td>
<td>Undo last action</td>
</tr>
</tbody>
</table>
Appendices

C. Materials Used in Global Control Strategies of Prolog Programmers Study

C.1. Standard Introduction

Thank you for participating in this experiment. All the data collected during this experiment will be treated anonymously. The experiment will last for approximately two hours. If you have any questions, do not hesitate to ask.

The aim of the experiment is to investigate the way in which experienced programmers, such as yourself, produce code. You will be presented with a short programming practice task, followed by the main task. For each task you will be asked to produce some Prolog code that fulfils the specifications which you will be given. During the tasks, can you please attempt to verbalise all your thoughts. Your thinking-aloud statements will be recorded in order to help with the investigation. Occasionally, you may be prompted to keep talking.

The experiment will be run on an Apple Macintosh computer, using LPA Prolog. A manual will be provided in case of any difficulties.
C.2. Specification of Practice Task

Practice Task
At the beginning of the experiment you will be given a short programming task. The purpose of this task is for you to familiarise yourself with the programming environment and to practice thinking aloud. When you are ready the supervisor will open the application. When the application is running, proceed to read the specification for this practice task on the next page. When you have finished, please inform the supervisor.

Specification for Practice Task
Produce a clause that reverses the contents of a list.
C.3. Specification of the Signals Problem

Main Task

You will now be given the main programming task. The purpose of this task is collect information on how experts produce code. Please, remember to verbalise all of your thoughts. When you are ready the supervisor will open the application. When the application is running, proceed to read the specification for this main task on the next page. When you have finished, please inform the supervisor.
Specification for Main Task

The Department of Transport has recently obtained a camera. The camera transmits ‘1’ after every ten minutes, and it transmits ‘2’ every time a vehicle passes it. During a day, the signals are collected into a data file. (The day starts at 12 am.). The DoT would like to know:

- The number of cars that pass the camera during the day.
- The number of cars for each hour of the day.
- The busiest 60 minutes during the day. The 60 minutes does not necessarily start on the hour.
- The lightest 60 minutes of traffic during the day. The 60 minutes does not necessarily start on the hour.
- The average number of cars that pass the camera in one hour.

Your task is to write a Prolog program. Using the day’s data file the program should produce a report that provides the above information. The report is to be channelled to the standard output, and should look like the example given below:

<table>
<thead>
<tr>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00am</td>
<td>3</td>
</tr>
<tr>
<td>1.00am</td>
<td>7</td>
</tr>
<tr>
<td>2.00am</td>
<td>15</td>
</tr>
<tr>
<td>3.00am</td>
<td>31</td>
</tr>
<tr>
<td>4.00am</td>
<td>49</td>
</tr>
<tr>
<td>5.00am</td>
<td>62</td>
</tr>
<tr>
<td>6.00am</td>
<td>77</td>
</tr>
<tr>
<td>7.00am</td>
<td>90</td>
</tr>
<tr>
<td>8.00am</td>
<td>112</td>
</tr>
<tr>
<td>9.00am</td>
<td>124</td>
</tr>
<tr>
<td>10.00am</td>
<td>128</td>
</tr>
<tr>
<td>11.00am</td>
<td>136</td>
</tr>
<tr>
<td>12.00pm</td>
<td>127</td>
</tr>
<tr>
<td>1.00pm</td>
<td>121</td>
</tr>
<tr>
<td>2.00pm</td>
<td>124</td>
</tr>
<tr>
<td>3.00pm</td>
<td>108</td>
</tr>
<tr>
<td>4.00pm</td>
<td>94</td>
</tr>
<tr>
<td>5.00pm</td>
<td>75</td>
</tr>
<tr>
<td>6.00pm</td>
<td>61</td>
</tr>
<tr>
<td>7.00pm</td>
<td>42</td>
</tr>
<tr>
<td>8.00pm</td>
<td>32</td>
</tr>
<tr>
<td>9.00pm</td>
<td>17</td>
</tr>
<tr>
<td>10.00pm</td>
<td>9</td>
</tr>
<tr>
<td>11.00pm</td>
<td>2</td>
</tr>
</tbody>
</table>

Total number of cars: 1646
The average number of cars per hour: 68.6
The busiest 60 minutes started at: 11.10am
The lightest 60 minutes started at: 10.50pm
D. Activity Transition Depths for Ss A1, A2 and A3

Figure D.1: A1's Depth of Activity Transitions

Figure D.2: A2's Depth of Activity Transitions
Figure D.3: A3's Depth of Activity Transitions
E. Materials Used in Global Control Strategies of C Programmers Study

E.1. Standard Introduction

Thank you for participating in this experiment. All the data collected during this experiment will be treated anonymously. The experiment will last for approximately two hours. If you have any questions, do not hesitate to ask.

The aim of the experiment is to investigate the way in which experienced programmers, such as yourself, produce code. You will be presented with a short programming practice task, followed by the main task. For each task you will be asked to produce some C code that fulfils the specifications which you will be given. During the tasks, can you please attempt to verbalise all your thoughts. Your thinking-aloud statements will be recorded in order to help with the investigation. Occasionally, you may be prompted to keep talking.

The experiment will be run on an Apple Macintosh computer, using Think C. This version is ANSI C compatible. A manual will be provided in case of any difficulties.
E.2. Specification of Practice Task

Practice Task
At the beginning of the experiment you will be given a short programming task. The purpose of this task is for you to familiarise yourself with the programming environment and to practice thinking aloud. When you are ready the supervisor will open the application. When the application is running, proceed to read the specification for this practice task on the next page. When you have finished, please inform the supervisor.

Specification for Practice Task

Produce a routine that writes out all the factors of the number with which the routine is presented.
E.3. Specification of Main Task

Main Task
You will now be given the main programming task. The purpose of this task is collect information on how experts produce code. Please, remember to verbalise all of your thoughts. When you are ready the supervisor will open the application. When the application is running, proceed to read the specification for this main task on the next page. When you have finished, please inform the supervisor.
Specification for Main Task

A local council intends to install a computer-based system that monitors the use of roads between any two junctions (junctions are numbered on the road map given below). The system will perform calculations on the data collected by vehicle detectors connected by data-links to a computer. Detectors are placed on each road, one for each direction that traffic can flow up a road. An example set of data for a day is provided in a binary file called ‘surveys’.

At the start of each survey a detector transmits a record of its location in the form of the junction number where the traffic comes from followed by the junction number it goes to. Whenever a vehicle passes the detector it transmits a signal consisting of the number 1. A clock in the detector is started at the beginning of the survey and at one second intervals thereafter it transmits a signal consisting of the number 2. At the end of a survey the detector transmits a 0. At the end of the final survey a second 0 is transmitted. Each signal is received by the computer as a single number (i.e. it is impossible for two signals to arrive at the same time).

Below is a plan of the roads, some of which are one-way whilst others allow traffic to flow in both directions.

```
\begin{center}
\begin{tikzpicture}[scale=0.5]
\t\node[shape=circle,draw,minimum size=0.5cm] (1) at (0,0) {1};
\t\node[shape=circle,draw,minimum size=0.5cm] (2) at (1,0) {2};
\t\node[shape=circle,draw,minimum size=0.5cm] (3) at (2,0) {3};
\t\node[shape=circle,draw,minimum size=0.5cm] (4) at (0,1) {4};
\t\node[shape=circle,draw,minimum size=0.5cm] (5) at (1,1) {5};
\t\node[shape=circle,draw,minimum size=0.5cm] (6) at (2,1) {6};
\t\node[shape=circle,draw,minimum size=0.5cm] (7) at (0,2) {7};
\t\node[shape=circle,draw,minimum size=0.5cm] (8) at (1,2) {8};
\t\node[shape=circle,draw,minimum size=0.5cm] (9) at (2,2) {9};
\t\draw (1) -- (2);
\t\draw (2) -- (3);
\t\draw (4) -- (5);
\t\draw (5) -- (6);
\t\draw (7) -- (8);
\t\draw (8) -- (9);
\end{tikzpicture}
\end{center}
```

Design a program which reads a set of signals and outputs the following for each survey:

(a) the rate of traffic flow (the number of vehicles per second);
(b) the length of the longest waiting period without a vehicle.

In the output the surveys will be sorted by the rate of traffic flow, and then by the length of the longest waiting period.
F. Occurrence of Opportunism

Figure F.1: Opportunism at Stages of B2's Design of Solution

Figure F.2: Opportunism at Stages of B4's Design of Solution
Figure F.3: Opportunism at Stages of B5's Design of Solution
G. Materials Used in Local Control Strategies Study

G.1. Introduction

In this experiment you will be presented with a small program written in pseudo code. The program reads in data obtained from a survey of traffic flow (vehicles per minute) along a particular road. The data is a list of average traffic flow for every 10 minutes. The purpose of the program is to perform calculations on this data.

However the four functions that perform the calculations have not yet been written. Your task is to complete the program by writing the four functions in pseudo code using a simple editor. You will be asked to think out aloud.

When you are ready I will present you with the program that is to be completed. The four functions to be written can be identified by the hash (#) next to their names. While comments in the program are enclosed by */ and /*. You are free to make any reasonable assumptions about the specifications.

The program should take about one hour to complete. Have you any questions?
G.2. Program To Be Completed

main_function()
begin
    read data into the array traffic_flow
    */ traffic_flow contains the rate of traffic flow
    every 10 mins starting at 6am */
    #wait(input traffic_flow, output length)
    */ wait returns the longest length of time when
    there is no traffic */
    #busiest_60min(input traffic_flow, output busiest_time)
    */ busiest_60min returns the start time (as text in
    12 hour clock, e.g. 4:40pm) of the busiest 60
    minutes of traffic */
    #common_flow(input traffic_flow, output median)
    */ common_flow returns the most common rate of
    traffic flow */
    #finish(input traffic_flow, output time_finished)
    */ finish returns the time (as text in 24 hour
    clock, e.g. 16:40) the survey finishes */
    display(input busiest_time, median, time_finished, length)
end
G.3. Questionnaire

Name .........................................................................................................................

How many years have you designed and implemented programs? ......................
.................................................................................................................................

What programming languages do you consider yourself to be fluent in? ..............
.................................................................................................................................

How did you decide in which order to complete the sub-routines/modules? ...........
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................

Please order the sub-routines in order of difficulty, starting with the easiest ........
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
.................................................................................................................................
H. Materials Used in Encouraging Use of Breadth-First

H.1. Standard Introduction

Thank you for participating in this experiment. All the data collected during this experiment will be treated confidentially. The experiment will last for approximately one hour. If you have any questions, do not hesitate to ask.

If for any reason you do not wish to continue with the experiment you are free to leave at any time.

The aim of the experiment is to investigate how the editor to be used in this study affects the development of programs. You will be presented with a programming task. For the task you will be asked use a particular editor to produce some C code that fulfils the specification given to you.

During the task, can you please verbalise all your thoughts. Your thinking-aloud statements will be recorded in order to help with the investigation. Occasionally, you may be prompted to keep talking.
H.2. The Editor (given to participants using the neutral editor)

The editor you will be asked to use is a simple text editor that has been adapted for a Breadth-First Strategy.

Many expert programmers have suggested that a Breadth-First Strategy is the optimum method for producing a program.

A program can be seen as a tree, with the main function at the root from which other functions branch. In a breadth-first strategy a program is produced at one level at a time.

![Diagram of a tree representing a program with levels A, B, and C.]

In the above picture there are three levels to the program. In a breadth-first strategy each level is completed before moving down a level.

The editor has a menu called 'Move' that lists all the functions you have mentioned in your code. When you have finished editing a function you have to select the next function to edit from the menu 'Move'.

The text editor will display the selected function, and all the functions that pass data to the selected function. The editor will only allow you to edit the selected function.

Finally, the editor beeps every minute: this is used to indicate the time on the tape recording of your statements.
H.3. **The Editor (given to participants using the supportive editor)**

The editor you will be asked to use is a simple text editor that has been adapted for a Breadth-First Strategy.

Many expert programmers have suggested that a Breadth-First Strategy is the optimum method for producing a program.

A program can be seen as a tree, with the main function at the root from which other functions branch. In a breadth-first strategy a program is produced at one level at a time.

In the above picture there are three levels to the program. In a breadth-first strategy each level is completed before moving down a level.

The editor has a menu called 'Move' that lists all the functions you have mentioned in your code. When you have finished editing a function you have to select the next function to edit from the menu 'Move'.

The text editor will display the selected function, and all the functions that pass data to the selected function. The editor will only allow you to edit the selected function.

The editor suggests functions (which conforms to the breadth-first strategy) for you to select by underlining those functions' name in the move menu.

Finally, the editor beeps every minute: this is used to indicate the time on the tape recording of your statements.
H.4. The Editor

The Editor (given to participants using the enforcing editor)

The editor you will be asked to use is a simple text editor that has been adapted for a Breadth-First Strategy.

Many expert programmers have suggested that a Breadth-First Strategy is the optimum method for producing a program.

A program can be seen as a tree, with the main function at the root from which other functions branch. In a breadth-first strategy a program is produced at one level at a time.

In the above picture there are three levels to the program. In a breadth-first strategy each level is completed before moving down a level.

The editor has a menu called 'Move' that lists all the functions you have mentioned in your code. When you have finished editing a function you have to select the next function to edit from the menu 'Move'.

The text editor will display the selected function, and all the functions that pass data to the selected function. The editor will only allow you to edit the selected function.

The editor only allows you to select functions that conform to the breadth-first strategy. When you move down a level, you will not be able to amend any functions at a higher level.

Finally, the editor beeps every minute: this is used to indicate the time on the tape recording of your statements.
H.5. The C language

In this version, the curly brackets cannot appear inside a function. For example the following is not possible:

```c
main()
{
    if (TRUE) {
        instruction1;
        instruction2
        instruction3
    }
}
```

You have to put the contents of the 'block' into a separate function. For example to achieve the above you can write:

```c
main()
{
    if (TRUE) function1();
}

function1()
{
    instruction1;
    instruction2;
    instruction3
}
```
H.6. Specification for Main Task

A local council intends to install a detector that monitors the use of a road. The data collected by the detector is stored in an array called 'survey'.

The detector starts its survey at 6am, and it can finish at any time. Whenever a vehicle passes the detector it stores the number 1. A clock in the detector is started at the beginning of the survey and at ten minute intervals thereafter it records the number 2. At the end of a survey the detector stores a 0. Therefore the array contains a series of 1's and 2's terminated with a 0.

Design a program which uses the data in the array 'survey' to output the following:

• the average traffic flow (the number of vehicles per minute);
• the start time (in a 12 hour clock) of the longest waiting period without a vehicle.

For example:

<table>
<thead>
<tr>
<th>Start of longest waiting period</th>
<th>Average Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.20am</td>
<td>10.25</td>
</tr>
</tbody>
</table>
H.7. Questionnaire

Course/year ........................................................................................................................................

How many years have you designed and implemented programs? ...............................................
.........................................................................................................................................................

What programming languages do you consider yourself to be fluent in? ........................................
.........................................................................................................................................................

How long have you been using C? .................................................................................................
.........................................................................................................................................................

What is the largest program you have written in any language (approx. no. of lines)? ...........
.........................................................................................................................................................

What is the largest program you have written in C (approx. no. of lines)? .............................
.........................................................................................................................................................
I. Example of Analysis of a Prolog Programmer (A4)

The following sections show an example of the stages of analysis applied to data produced by each participant in the Prolog study and in the C study. First, the coded verbal reports of A4 are shown. Section 1.2. displays the final program produced by A4. Section 1.3. shows the goal hierarchy produced from the verbal reports and the produced program. Each node in the hierarchy contains a list of actions performed on the node. The actions are identified by numbers, and the numbers appear in the verbal protocols. Section 1.4. contains an analysis of transitions between nodes of the goal hierarchy. The order in which the nodes are visited (i.e. transitions) is determined by the order in which actions are performed on the nodes, as described by the action numbers. The analysis calculates whether each transition conforms to a global control strategy, from which overall conformance to a global control strategy is calculated.

1.1. Final Verbal & Keystroke Protocol of A4

The protocols consist of four columns. The third column contains the statements produced by the programmer. Some of the statements refer to more than one action type (c.f. Section B.1.), these statements are separated into parts by '/', where each part only refers to one action type.

The first column contains the time when an action occurs. Time is measured in minutes from the start of task. The second column is the action type of the utterances in the third column. The fourth column enumerates actions performed on the goal hierarchy. Since the table is also derived from the keystroke protocols, the action type F1 (writing code) is not always accompanied by an utterance.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action Type</th>
<th>Verbal Report</th>
<th>Action Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td></td>
<td>The Department of Transport has a camera, it transmit a 1 after ten minutes, 2 every time a vehicle passes it.</td>
<td></td>
</tr>
<tr>
<td>G1</td>
<td></td>
<td>During the day, the signals are collected into a data file.</td>
<td></td>
</tr>
<tr>
<td>G1</td>
<td></td>
<td>The day starts at 12am on Monday.</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td>Yes.</td>
<td></td>
</tr>
<tr>
<td>G1</td>
<td></td>
<td>Calculate the number of cars that passes the camera during the day, not maths. oh dear, I can’t do maths.</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td>I thought I was suppose to do computing, not maths.</td>
<td></td>
</tr>
</tbody>
</table>
The busiest 60 minutes, the lightest 60 minutes, the average number of cars that passes in an hour.

Using the day's data file the program should produce a report.

The task is to write a Prolog program, using the day's data file the program should produce a report providing the above information.

The report should be channelled to the standard output, and should look like the example below.

So, hang on.

Do I need to actually, so I need to actually define the format of the data file that's collected?

[No]

Hang on, so it transmits a 1 every ten minutes, and then, so it would be 1 1 1, and every time a car goes through within that 10 minutes then it puts a 2, there's no information about when the 2 happens, I just know that it happens within that 10 minutes. Okey-doke.

And it's a number on each line, is it?

[Yes]

So, what do I want to do, during the day, so.

The first decision of all, I think is I've to decide, the first thing I would have to decided is whether I was going to read in the data file / and get it stored internally, in it's entirety, / or whether I was going to try to do the processing on the fly as I read in the data file.

And it will probably depend on the nature of the data file as to which I would go.

If the data file isn't that long I would probably be tempted to read it all in, / into some internal Prolog facts, or some thing like that, / and then do the manipulation as a, from the Prolog fact base.

Otherwise, I would have to read things in and try / and calculate these things on the fly / and some of them might not be able to calculate on the fly or you know, um, as your accumulator.

The busiest 60 minutes during the day.

[pause]
<p>| D2 | For some thing like this, I mean the first, I think actually the way I would probably do it is a first parses is to load in the data file / as Prolog facts / and, er, and work from there to get the ideas sorted out, because, you know, just sort of going of, trying, you know, do it as you go for, something like the busiest 60 minutes and the lightest 60 minutes, sort of fills like you need everything there to be able to work out which is the best. |
| H | It's the type of thing where you might do a prototype, where you load everything in the first time and then do an analysis, to get your programming ideas sorted out and then you might reimplement it with a better algorithm in mind where you just sort of do all the things, accumulate what you need as you go along and then you don't have to worry about, you know, if you suddenly get to, the thing put on a every busy road that, loading up 20,000 cars that have been through during the day blows your program out because you don't have the memory. |
| H | Okay, so lets start going, lets assume that you going to read everything in first, that's the easy way, isn't it, lets go the easy way first, get the ideas sorted out, that's what my mum says, &quot;get yourself sorted out, boy&quot;. |
| A1 | Right okay, so the first thing I'm going to read in is the file, so. |
| A2 | Let's have a completely unique and original name for the top thing, go which is going to do something. |
| H | So we are going to read_in_datafile, / and then we are going to analyse the data, / and then we are going to process the data, no then we are going to report, generate_report. |
| C | Now, do I actually analyse the data / and pass the variable from the data into the report / and generate the report / or whether I assert the data into the, sort of, Prolog fact base / and generate the report from that and, I might do that first and then just see how it goes / and decide whether I'm going to pass it / as an argument, as a data structure, / or actually just use the Prolog fact base as data structure. |
| D3 | It's nicer to pass it as an argument but again just. |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>So read_in_datafile, file, / um, do I assume I always know the name of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2</td>
<td>this data file or prompt?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[Yes - survey]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>It’s always survey, okay so I can hard code that in, yuck, urk, urk.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So I want to, oh, this is where we have fun and games with syntax, survey.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>So now I want to get a line, / um, no I now want to read_in_datafile / and what I’m going to do because I’ve just got a list of 1’s and 2’s, I have got to do my own in effect time-stamping, so what I want to do is I want to read in, / I to make want a recursive thing / that will read in a line at a time / and if / the line is 1, / then I’ll bump up my time-stamp thing, / and if / it is a 2, / I’ll just assert it with the current time-stamp and just work like that.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So, read_in_datafile / with the initial time-stamp of 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>So, I’m going to be asserting what I, / I’m just going to read the data file in / and assert it in the database, time-stamped.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Um, so I’m going to read in the file and I’m going to clause it, / and close survey.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Keep open and close in the same top level predicate just so, otherwise you can get lost as to where, what files are open and where it closes, it just a nice style to keep these things together.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>So, read_in_datafile, / TimeStamp is the first variable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Right, so now I’m going to read in line, there is a read in line, isn’t there?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2</td>
<td>[Me-get]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2</td>
<td>Get, get, are you sure?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2</td>
<td>Is there a newline, or is there just a constant string of 1's and 2's?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>---------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[newline, get ignores newline]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>That's true, so we are going to get, um, does LPA have the notation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>were you can have 0 dash for octal?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[don't know]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Okay, I do it the way I would normal do it and then sort of comment</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out, and do it some other way and put comments around.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get a Ch, um, so right, / if Ch is identical to, now normal in Quintus</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>notation and I think it is going to be in standard Prolog, LPA is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>suppose to be Quintus compatible, maybe just check, just, compiler</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>will tell us.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>Um, that notation will be, when that, when Prolog has compiled that</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>notation it just converts it to character code, but it is just more readable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>You know it's character 1 whatever system you go on to, that will be</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interpreted to whatever character code is appropriate for that</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>architecture.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>If it is character code 1 / then I want to bump up new time stamp,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NewTime is Time, I'm just going to do TimeStamp is, um, you know,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 is after 10 minutes, 2 is after 20 minutes, just er, again, I suppose, I</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>suppose there is no problem doing it in, no lets do it in 10, because if</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>they change their minds, they want to, you know, to increase the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>resolution then it is less hassle for the program to adjust later, so new</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>time-stamp is plus 10.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1</td>
<td>There's one every 10 minutes, arh, is there one initially when it starts?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2</td>
<td>[no - after every 10 minutes]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>So, okay, so move that up to 10, that's fine.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>New time stamp is that, otherwise, I'm using a Prolog, this Prolog if-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>then-else thing I do it as, a sort of, separate clauses and things like that,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>which will be probably just as, may be as just slightly less efficient,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>um, depends on whether its got first argument indexing or not.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>Shall I, no lets do it this way first, this is like having a local cut, but it's</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>on most Prologs, it's sort of, you know, you not really losing</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>anything, because I not choosing to use the local cut, it's just a test, so</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>there's no ambiguity there, so I'm not losing any logic there at all.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>Otherwise if / it is 2 / then assert, a hideous crime, assert, um, yes, I want to assert, it’s a vehicle not a car, vehicle at the current time-stamp, so I assert a fact, urk.</td>
<td>29a</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>H F1</td>
<td>Assert vehicle at TimeStamp, / otherwise / it’s an error.</td>
<td>31a</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>Right, [writing Ch == -1] then true to succeed.</td>
<td>31b</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Ah, right so we’ve got to recurse.</td>
<td>33a</td>
<td></td>
</tr>
<tr>
<td>13 C</td>
<td>Okay this is where I’m going to use this, um, this style, where I’m going to test and recurse.</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>May be it is better, okay, this is where it is difficult, on this type of exercise there, you know, you I would do it and how I would do it nicely, and how I would do it, if I’m being really good about this then I would actually create separate clauses for each of these instances, um, because then the recursion would be more obvious so it might be easier for someone to read but personally if I writing this for myself I would do the recursion.</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Let’s do this, LPA is not very nice for doing indentation, is it?</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Right, recurse through and this is where I bet that LPA’s compiler is not as smart as Quintus in optimising, but it might be; NewTime.</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>So basically if / we get a 1 / we bump up the time stamp by 10 / and then recurse with that new time-stamp.</td>
<td>35a</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>Otherwise / we assert, / if it is a 2 / we assert if we had a vehicle at that time, at that current TimeStamp / and recurse with the same time-stamp.</td>
<td>36a</td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>Or, / it’s the undefined / in which case we, we, um, just succeed and don’t anything, we’ve finished, we not recurse.</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>B4</td>
<td>I’ll put another, I probably but another case in here, where it is not any of these, in which case having these tests as local cuts actually safes having cuts, to catch; if I don’t want to do the trapping of errors then easy, then I’ll probably wouldn’t have to worry about local cuts and doing this another way, I could use first argument indexing and be sure that it was just as deterministic, but if I want to do a catch all type of thing, if it is none of these then it probably more efficient using this type of if-then-else structure anyway.</td>
<td>43</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>F1</td>
<td>So, otherwise, um, lets just write, how do you write stuff the terminal, write unrecognised input, um, / and put whatever it was there, so we know that it was unrecognised, / and lets just on recursing anyway just be friendly, just in case it was a just fluke, so we can still process the data.</td>
<td>44</td>
<td>44a</td>
</tr>
<tr>
<td>C</td>
<td>We could just bomb out and say this is a dude data file, but it is probably more helpful to just continue and assume that it was some noise that got in some how.</td>
<td>45a</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>But at least they have been alerted to there being some dodgy thing in there; with the current TimeStamp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>And that is the end of that thing there, so recursing there, read all now, we’ve got a whole pile of facts saying vehicle at a particular time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>A1</td>
<td>Um, so that we’ve got that data it is easy we can go on to analysis.</td>
<td>46</td>
</tr>
<tr>
<td>G1</td>
<td>And, the number of cars that passes during the day, so the total number of cars.</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>The number of cars for each hour of the day.</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I</td>
<td>Okay.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[pause]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>I’m trying to think which, probably the way that I would approach it first, is to just get it going and then to work out, you know, where there’s, where I can simplify it later on, because once, now I’ve just got those things as facts, to just to start on the ones that occur every hour, because, they are quite simple, um.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>This is where I can see already that I could be doing so of these things as I was reading it in, instead of parsing the same data over several times.</td>
<td>48a</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>To bad, I’m going to parse over the data several times [mumble] and maybe come back and optimise it later.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>That’s the worst thing with sitting in front of a screen and trying to solve the problem at the same time, the temptation is to just get something going.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1 A3</td>
<td>Right, so analyse, analyse_data.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3 G3</td>
<td>So first of all you want to put out the number of cars for each hour / and then the total number, um, what is the best way of doing this?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 D3</td>
<td>Incrementing, the facts there, I’m sure I will be better of doing this as I was reading this in, actually, these first ones.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G1</td>
<td>Um, maybe I’ll do it as it comes in.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Trying now to collect all this up to report it, I’ve got the stuff there, so I need to, / every time I need to start from 0, / and every time the, / sort of count everything up until I get to something mod 60 / then I’ll output the total so far as that was an hour’s worth, / put it into the overall accumulator for the total that’s gone, / and then start collecting up for the next hour.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>And just every mod 60, because I’m keeping to the minutes thing, / and putting it out, um, I could do that on reading in, / and then I got the average at the end of that.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>It’s just this busiest and lightest thing that bothers me.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>I’ll have to think about how to do that.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>The average number of cars that pass the car in one hour, that’s the average, that’s just the total number out, um, perhaps I will go back and do this as I come in, that might be, that might be easier for this first bit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>Now I might not actually need to assert these things after all, I want to assert them for the moment just for the time being.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>I’ve got to think how to do the busiest bit first.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4 C</td>
<td>As I’m reading stuff in, / what I’m going to need, I’m going to need the current time-stamp, / we need the, um, an accumulator for the current hour, / and an accumulator for the current day, and they are going to be just numerical things that I bump up as I go down, I’ll test those, depending on what.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>I’m going to need to know extra thing, I need to know the current hour, let’s call it HourTotal and DayTotal.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>So these are really just like variable that are just going to be incremented as I go down.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Right, so when I get something and decide what I want to do with it, I need to do testing on when I put stuff out.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Um, so basically I’m going to start reading in, / and if, / every time I get to 60 minutes, the current hour mod 60 / then print out what I’ve got so far / and start again, so I’ve actually got to get this 60 minute one through to know I’ve got to print out what I’ve got so far, / so I have to actually read in the, sort of, new time-stamp stuff before I can decide, you know, whether I need to print it out.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>So when I get a time-stamp thing that is when I need to do the test, I’ve got the NewTime, so I want to check whether I’ve report it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>Try_report_hour, / increase NewHourTotal at this time, which will be NewTime, / new count hour, / new time, then I’ve got time-stamp, well I know the time, I’ve been a prat here, so, um, I’ve just deleted an argument there I realised I already had it.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>So I want to test the new time and depending on what it is, then I want to print out the hour total.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>When I have got to the end of file now, / that is when I’ll report / the total number / and average, report, report_day.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>And I assume that there are 12 hours in a day, that’s going to be an invariant isn’t it?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>No 24 hours, so I don’t need to worry about coding that.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So now when, okay, I’m going to forget asserting for the moment, I’ve just commented it out, I haven’t trashed it I might still want to assert it to do the busiest hour.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>I’ve commented that out for the moment, so I remember it’s there.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So whenever we get a vehicle / we need to bump up the HourTotal, um, [mumble], that’s bumped up the HourTotal, NewHourTotal, HourTotal plus 1.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[tape break]
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D1</td>
<td>So, I've now got a new argument, new total hour / and I also need to bump up the day total, no, I bump the day total every hour, so I keep the hourly total, then that way means that I only have do that arithmetic every 6 cycles rather cycle, so I'm saving an arithmetic calculation there, so keep the same day total for the moment and increment the day total every hour, so that one, that one, and this one, this one.</td>
<td>74a</td>
<td>75</td>
</tr>
<tr>
<td>C</td>
<td>F1</td>
<td>The unrecognised input one, / where I'm just going to recurse and do nothing, just past basically the same input as output, doesn't touch it.</td>
<td>75a</td>
<td>75b</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>So this first one where I've actually bumped up the time-stamp, / um, as well as trying to report the hour, what do I want to do, I also, / if / it is an hour / I want to bump up the day count, so I'm going to try to change that try_report_hour to actually test if it is hour and going to chunk out the total so far and I'm also going to increment the day total.</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td>So I'm going to need to pass in DayTotal to it, / and get NewDayTotal, NewDayTotal out.</td>
<td>80</td>
<td>81</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>That's the problem with these long meaningful variable names, they're to big.</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td></td>
<td>Um, try report, NewDayTotal, so read_in_datafile, / NewTime, um, / I'm going to need NewHourTotal as well, ain't I, because it could be a new hour, then again it might not, NewHourTotal, / NewDayTotal.</td>
<td>83</td>
<td>84</td>
</tr>
<tr>
<td>E2</td>
<td></td>
<td>Right, so read in character, / if / it a 1 / we are going to bump up the time-stamp by 10, / and then we are going to check whether we hit an hour spot.</td>
<td>85</td>
<td>86</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>If / we've hit an hour spot / then we are going to print out the total to date / and give a new hour total of zero / and but if, I haven't written that bit yet, but if, if it is not a hourly thing, / then the new hour total is just going to be the same as the previous hour total, / and similarly if it is an hour then the day total will be incremented / and then I will out those NewHourTotal / and NewDayTotal recursively, right.</td>
<td>88a</td>
<td>89</td>
</tr>
<tr>
<td>30</td>
<td>A3</td>
<td>So now I just want to write a try_report_hour, lets call it check_report_hour.</td>
<td>98</td>
<td></td>
</tr>
</tbody>
</table>
Right, so test, um, so if NewTime [looks at manual] 99
Equality test, that was what I was after. 100
So NewTime mod 60, um, evaluates to 0, if it does then we’ve hit an
hour. / so now what do we want to write out the current hour’s total.
So, now I have got translate the minutes to this format, haven’t I,
okay. 101
Write_hour, so I just pass it, um, it’s going to be NewTime / minus 10
isn’t it, because that is the previous time. 102
[mumble]
Let’s do it this way.
So I worry about formatting the hour later, I’ve got it in minutes at the
moment, I’ll just, sort of, write a thing that format the minutes into
that, sort of, hour, AM, PM thing later.
So write the NewTime, I’m sure they got a pretty format output thing
as well, but to bad.
That’s just a couple of spaces, not justified or anything, is it?
Right, and then the HourTotal, HourTotal, and then what do I want to
do?
I want to make the NewHourTotal would be 0.
And the NewDayTotal is the current DayTotal plus this HourTotal, I’m
just incrementing the DayTotal every hour, I could have done it every
10 minutes, but this saves that extra number of computations.
Otherwise, what have we got, otherwise we don’t want to do anything,
we want just to pass every thing through, slip through.
So NewHourTotal is HourTotal, / and NewDayTotal is DayTotal,
that’s the end of that, I think.
So, um I can do all the other calculations at the end of the day, / lets
just do a quick thing, I think I’ll do this write_hour.
So I have got something coming in Mins / and I want to format that
into AM and PM time, / but that’s, that’s also, that’s the next hour
marker, so when I get to, when I get to 60 minutes that’s really, um, to
make that 0 hours.
Um, I coming on hours, aren’t we so I can just do that.
Div 60, so Hour is Mins, wants div, div 60, I want minus 10, / I
suppose I could do minus 1, makes no difference.
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>And then I want to write it out in that format.</td>
<td>117</td>
</tr>
<tr>
<td>H</td>
<td>If they decide they want to report on a higher resolution, they will have to just change this bit.</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Um, so I want to write, I want to write.</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>I also need to know whether it’s greater than or less than 12.</td>
<td>118</td>
</tr>
<tr>
<td>A1</td>
<td>Um, what a pain that format.</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>I could do a couple of calculations, or a could do a test and a calculation.</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>A test and a calculation is actually, probably faster than doing two alpha-integer calculations, but whether it is confusing or not, I don’t know.</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Because what I could do, is just say if the Mins is above the number of minutes for midday then do the calculation div 60 minus, um, 13 to cope for, taking the morning hours off, otherwise just do a minus 1.</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>So lets do it that way, lets do the intelligent, efficient way, since I write lovely efficient unreadable programs.</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>So, if Mins, no because we really need to put it nicely.</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>I have to, I suppose I could sit and calculate the number of Mins, you know, if I had a calculator here, or have I, yes I’ve got a Mac.</td>
<td>119</td>
</tr>
<tr>
<td>C</td>
<td>I could quickly work out, you know, what the number of minutes is for it to be above midday and program that in as an integer, so that the actual programming would be, then I would be testing against a prefix integer, predefined integer and then doing a calculation which I said is faster than do two calculation because I could do a calculation to get the number, hour number and then if it is greater than 12 do another calculation to subtract 12, maybe that’s more readable.</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>It means half the day you’re doing two calculations, whereas the other way you’re always doing, you doing, you always do one calculation and one test, and sometimes you’re doing two calculations and a test.</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>The other way you’re testing before you’ve calculate, I mean there is always one calculate, one test and one calculate.</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>But if you ever change it, you’re not going to change it, it is always going to be fixed midday, it’s mixed isn’t it?</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>It just, you know, for me to put a predefined integer in there, I could put an integer in there, but I’ll have to put a comment that says this really equates to the number of minutes to midday.</td>
<td></td>
</tr>
</tbody>
</table>
Just so that whoever looks at this afterwards knows where I've conjured this number from, but that, otherwise I could put in here to calculate, you know, as part of the thing, on the fly, the number of hours per day, so it is easy to see someone would see, but, you know, it sorts off defeat the whole object.

So, let's do it this way.

Mins is greater than, no sod this, let's do 2 calculations, we're just playing rough and ready, aren't we?

So let's do a test, rewind the tape.

Hour, if / Hour is greater than, greater than 12 / then / UseHour is Hour minus 12, / otherwise / UseHour is just Hour, / right so now I just write UseHour.

And then another pretty printing stuff.

Write, and then write out, um, no hang on, UseHour and then I want to write out dot 00, something.

I've got this test up here, I need to do.

If UseHour is, UseHour, I have to do an extra thing, past a flag of, past a flag set to whether it is going to be AM or PM, and AMPM is PM.

If UseHour is Hour then AMPM is AM.

Right, just the 00 minutes / and then I want to write out AM, I want to write out, no I did that all before, didn't I?

I just need to write out the hour, so that should print out the time of it, I hope.

I should just probably put a comment here saying we're doing minus 1 so that, um, we've already hit next year.

Just, so, you know, it is obvious why we're dividing by 60 to find the hour, but why I'm taking 1 off, um, to report, it might not be clear in the code, it might be, you know, they might have forgotten by the time they looking at this, so just as a reminder that we don't report it until we've actually hit the next hour.
So the current hour is under the current minute selling is 10 minutes ahead.

No this is wrong isn’t it, this is bullocks.

I do need to do minus 10, no hang on, no, no, no.

If I’m doing div 60 there be 10 minutes left over, it doesn’t matter, I just need to do the div 60.

So there are 10 minutes extra.

So what happen is, I’m processing from 0 up to 60, hang on, I was right, wasn’t I?

What do I want to do, processing 0 up to 60, I’ve hit 60 so I know I’m on the next hour but I want to do everything up to then.

I was right, I was right, "Trust your instincts, Luke. Follow the Force."

That’s right, so we’ve done the test before, we just want to find out, um, when it is an hour, so every 60 minutes print out, so we know div 60 is going to give us an integer, and we want to, right, we’re there.

Right, okay, I know I’ve done that bit.

Right, so now that I’ve done that bit, I need to do the writing out, um, the report day business.

Report_day, / DayTotal, um, so I just want to write, I didn’t do any newlines and stuff like that, did I?

Write HourTotal, / comma, lets do a newline, it does help.

So I want to do an extra newline here.

Write out the text, / total number of cars, write DayTotal, / newline, newline, / and then the average / I know, the average car per day is just that divided by 12.

So Average.

Average number of cars per hour, no hang on.

Does that mean the average of the average number of cars or the total of the average?

The same isn’t it, it’s probably be the same.

The Average is DayTotal divided by, we’ve got lots of real precision there, so we don’t want a integer.
How can you have average number of cars, you’ve got half a car going pass, it was going so fast we didn’t see it.

Um, divided by, Average is DayTotal divided by 24, / we need to write, the average number of cars per hour, / Average, / newline.

So we’ve finished reported on the day there, so hopefully that’s the easy bit done.

Do we test it now?

I suppose I should, shouldn’t before I go on the hard bit, to make sure that this works.

No, would I normally, no I would just plough on and test it right at the end.

Um, I think I will go back and change this, the top level thing, so I read_in_datafile, I analyse much of it on the fly.

I need to change these things, so we start off, just to start off properly.

[change the parameters to read_in_datafile so that match the definition]

Um, right I’m fairly confident that that does those first bits, so the busiest 60 minutes and the lightest 60 minutes, this is where I go and make a coffee and move away from the screen and make about how I’m going to solve this before I just plough in to it.

So we’ve got 10 minutes intervals, so we’ve got a sliding 10 minutes.

So what we can do is as we’re reading in, keep a 60 minute window total, / and as we read stuff in we slide that down and if at any point that 60 minute window / adds up to more than the current biggest / or smaller than the lightest then, / um, I can, er, sort of, put a new lightest / or a new heaviest.

So this is going to mean, loads of lovely, I could do this on the fly still of course as I’m reading in, because I could just keep a window there, the only thing I’ve got to be careful of is when I start off, I can’t start, I suppose I just do it every hour don’t I, no it’s a sliding, maybe start on the hour, it’s a sliding 10 minutes, so I’ve got to be careful when I start off is that I don’t have a 60 minute window yet, so I can’t actually start testing my window until I’m an hour in.

So I just got to make sure as I’m building up this sliding window that until I have gone past the first hour then, / I don’t try to do anything.
<table>
<thead>
<tr>
<th>I</th>
<th>So want I will probably do, because I'm keeping a sliding window.</th>
</tr>
</thead>
<tbody>
<tr>
<td>G3</td>
<td>How would I do this?</td>
</tr>
<tr>
<td>C</td>
<td>What I really want to do is just keep; the obvious way to think of it is I've got a window of 6 10 minute slots, so I could just have a list of 6 items with, you know, each element in the list being a 10 minute slot, and as I slide the window down popping one off the front and putting a new one on the tail.</td>
</tr>
<tr>
<td>D1</td>
<td>Popping of the front is quite efficient, but putting a new one on the tail is pretty inefficient if I'm sliding the others along at the same time.</td>
</tr>
<tr>
<td>C</td>
<td>I mean, I could probably, it's not to bad because if I got a fix, if I know the length, the window is always going to be 60, then I could quite easily do it just with variables with, you know, with something that, you know, with a 6, 6 element list, swapping, shifting 6 along, each one in the window isn't to bad.</td>
</tr>
<tr>
<td>D1</td>
<td>If suddenly, you know, if the resolution was higher then and suddenly we had a larger window it might become more painful, but lets stick to this route, probably what I would do is, um, have a current window which was a list of 6 elements.</td>
</tr>
<tr>
<td>C</td>
<td>Something like X1 to X6, and then I would have something like a clause that was, um, a new window / and what you would do is pass in the current window as the first argument say, / ignoring the head of that, the first element of that list, because now that has slide off the window, so I would probably to make the reading clear, I would start the first element of the window list would be the oldest part of the window.</td>
</tr>
<tr>
<td>C</td>
<td>So I would, if you imagine the hours as going linear from left to right, and the sliding window, the list is going left to right and sliding along with it.</td>
</tr>
<tr>
<td>C</td>
<td>So, then I would be have X2, I would have a new X1 to a new X5, / and then I would the new time-stamp total, hang on, hang on, hang on.</td>
</tr>
<tr>
<td>G3</td>
<td>Do I record, I don't record when the time-stamp changes do I?</td>
</tr>
<tr>
<td>B5</td>
<td>I'm aware of it there, so each time that changes that is when I would, I've got an hour total there, so I've got, that's an hour total.</td>
</tr>
<tr>
<td>B4</td>
<td>I need to have an accumulator for that current time slot as well, don't I, if I'm doing this on the fly.</td>
</tr>
</tbody>
</table>

---

-214-
Thousands of arguments, it’s doing it on the fly, do it this way is probably, you know, it is feasible this way, so I will avoid using assert doing it on the fly and like I said it doesn’t matter how big the data file is.

It is only going to be what a maximum of, in the worst case is where there is going to be 10 arguments which isn’t outrageous for Prolog, because you don’t have global variables.

Treat the arguments as equivalent to global variables, pass them around.

I will still do it this way, yeah, so lets stick with that.

So we will have new 10 minute total, so we are going to have a 10 minute, you know, time-stamp accumulator, as well.

Um, have a TimeTotal coming in, and every time, when would I change this?

I probably change where I do the bumping up.

Lets get this idea sorted out first.

I have a NewTimeTotal then I’d output a new window, X1, and that would now be X6, 1 to X6.

So that would be how, I just call that / with the old window variable, / the new time slot total / and then, um, get the, the new window out, / and then every change in time slot, so I’ll check the report on the hour / and at the same point I would check on the time window there, okay so lets try that, lets go for it.

Um, so I said I need a new accumulator for the name thing which is TimeTotal.

I have got to make sure all my places where I recurse actually remember to keep the right number of arguments, that’s a classic, when you compile it suddenly tells you that there are things of different arity or not together, which normally means you’ve forgotten to add the variable in, in one of the recursive situations.

So I’m going to have a new time total there, NewTimeTotal; and I’m going to have a new time total or am I going to have time total; NewTimeTotal is always going to be 0, so it is always going to be 0.

So what do I want to do, I was thinking whether, where I do all these bumping up of totals.
So I really want to know, where I read a car then I want to bump up
TimeTotal, NewTimeTotal rather than the HourTotal, NewTimeTotal is
TimeTotal plus 1.

So it is not a NewHourTotal bumped up every time I get a car, so I
now I'm bumping up every time a new car into the TimeTotal, then
every time I get a new car in segment that is when I can bump up the
HourTotal, there.

So where shall I do that?

Do I need to do that?

By testing where I check, um.

So now as well as checking for the report hour, / I'm going to always
check for, um, the heaviest and lightest as well, as well as a current, as
a sort of current accumulator, um, let's keep that as a single argument
shall we, for the moment.

So let's have a single argument, I just make it a list of two, / the light /
and heavy later.

I mean it is not so efficient, you know, I could have two arguments
lightest and heaviest in terms of loads past around everywhere, but it
means every time I recurse I've got to have those two arguments in
there and the only place I need actually collect them is going to be
within this sub, one part of this thing that it might be easier to break it
out there at that point and so to avoid to have to pass round two
variables constantly.

It is no so efficient for execution, Prolog takes slightly longer to pass a
more complex argument than a single argument, but it just means that
there is not so much going on at the higher level, it is a bit more
readable, perhaps.

So check_load_window, let's call it window, just to confuse people.

Check_load_window, I'm going to have current window aren't I as
well.

Let's have an argument which is the LoadWindow, that is going to a
complex thing, / that's going to have the max, / min, / CurrentWindow
as a complex structure.
So we are going to have check_load_window, / LoadWindow / with TimeTotal / and get a NewLoadWindow.

So NewTimeTotal, / NewDayTotal, / NewLoadWindow.

For these other ones, um, LoadWindow will stay we same.

This clause is getting slightly big and cumbersome now, but I'm not considering breaking it out until, sub-sub-predicates, it's not that bad, but, especially on this editor it looks a bit unreadable.

It's starting to look a bit heavy, I might break it down into separate cases, separate clauses to just try to make it a bit more readable.

Lets keep on ploughing on like this for the moment.

Um, check_load_window, / LoadWindow, [mumble], / LoadWindow, / DayTotal, oh hang on, put in DayTotal / and TimeTotal.

Right so check_report_hour, now I've got to sort out what I'm doing with keeping the HourTotal, now I've gone to bumping up the TimeTotal every time I hit a car, so every car, NewTimeTotal, / so every time I get a new time, then I want to add the NewTimeTotal on to the current HourTotal, and I always want to do that don't I?

Before I actually report on it, because it's part of the past.

So the first thing I want to do, pass in TimeTotal here, I always want to do it, don't I, and then I need to there, yeah.

No I suppose, hang on, do I need the TimeTotal accumulator, no I just want the 10 minutes one there, so I don't need the HourTotal.

I've got TimeTotal, bump up HourTotal, the HourTotal is [?], right okay.

So, um, put TimeTotal here.
Some people might balk at having lots of variables, and probably if I was doing this for myself I might actually truncate some of these variable names to make them less readable, but then it won't be so big and long and wrapping round, but um, since I'm being a good boy, I'm giving them nice meaningful names.

<table>
<thead>
<tr>
<th>H</th>
<th>Some people might bulk at having lots of variables, and probably if I was doing this for myself I might actually truncate some of these variable names to make them less readable, but then it won't be so big and long and wrapping round, but um, since I'm being a good boy, I'm giving them nice meaningful names.</th>
<th>191</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Um, so TimeTotal, so now I always bump up the HourTotal with the TimeTotal.</td>
<td>192</td>
</tr>
<tr>
<td>C</td>
<td>So new HourTotal is 0, / NewDayTotal is, do I need HourTotal?</td>
<td>193</td>
</tr>
<tr>
<td>G3</td>
<td>I do, don't I?</td>
<td>[mumble]</td>
</tr>
<tr>
<td>B2</td>
<td>I'm going to have to introduce a couple of different variables here now.</td>
<td>194</td>
</tr>
<tr>
<td>C</td>
<td>Because every time I get, TimeTotal, every time I get a new time, / I've got to add the NewTimeTotal to the HourTotal, then I've got to do different things with that.</td>
<td>195</td>
</tr>
<tr>
<td>F1</td>
<td>So, sort off, LastHourTotal is the current total plus the TimeTotal.</td>
<td>196</td>
</tr>
<tr>
<td>I</td>
<td>And so, then.</td>
<td>[pause]</td>
</tr>
<tr>
<td>G3</td>
<td>What do I want to do here?</td>
<td>197</td>
</tr>
<tr>
<td>E2</td>
<td>I've got my HourTotal which is the hour total so far, / so I've just got NewHourTotal, this is what comes from changing my, perhaps it is just NewHourTotal after, I have to think about this.</td>
<td>198</td>
</tr>
<tr>
<td>C</td>
<td>NewHourTotal is the HourTotal plus TimeTotal, yeah, always, no it is not always.</td>
<td>199</td>
</tr>
<tr>
<td>C</td>
<td>LastHourTotal is HourTotal plus TimeTotal, right, um, / NewHourTotal is always 0 when we've hit a 60, um, / so I want to add LastHourTotal on to DayTotal now instead of the previous thing I had, so LastHourTotal is that now.</td>
<td>200</td>
</tr>
<tr>
<td>F1</td>
<td>Now, here NewHourTotal is LastHourTotal, / NewDayTotal is that, right that makes sense, I always, I'm just it in here since I'm passing in here I could do it at the top I suppose.</td>
<td>201</td>
</tr>
<tr>
<td>70</td>
<td>I probably could do it at the top, it might be less confusing with having all these blumy hour totals around, rather than having it down here.</td>
<td>202</td>
</tr>
<tr>
<td>C</td>
<td>TimeTotal, / HourTotal, LastHourTotal come in here.</td>
<td>203</td>
</tr>
<tr>
<td>A3</td>
<td>Don't need the TimeTotal any other reason, er, no I'm going to move it back up here.</td>
<td>204</td>
</tr>
</tbody>
</table>
Change, change of plan, because it's not necessary to have it down there, it saves having an extra argument in that thing, really.

Where do you put that extra line, Paul?

In here, so then, I'm going to sticking LastHourTotal in there, right.

So, this is, and I think everything else should be there, should be okay, yes.

I've changed that there already, / so I really just moved the calculation up to save an argument, / and check_report_hour it is not necessary to have the 10 minute total in there so why pass it down?

Good question, but I need it in check_load_window, / let's do check_load_window now.

[?], still got that gash stuff, not doing that way any more, I'm just going to read in data file.

So, go suddenly becomes trivial, it suddenly becomes read_in_datafile.

1 2 3 4 5, oh I need my empty window, I don't know what that is going to be like yet, so I just put in a place holder to remind myself, sort of, to put something in there later.

Right so check my window, just position the predicate so that it, sort of close to the other one that it's calling.

It is always difficult to know where to, how to group predicates, I tend to group them in the order they are called so you start off with the top and keeping the main predicates at the top, and any subsidiary predicates just sort of have them in the same order but lower down keeping the nice periphery sort of ones to end.

So, check_load_window, right, so this is where I need have to decide about how it is all to be represented.

Um, so what I want to do, so my LoadWindow is going to be a sort of complex structure, I could be untidy and easy and just make it a list containing 3 things, / where one of the list is the window, / the other thing is the Max, and the current max, / and current min, / and, but it is actually more efficient to have a term with three arguments than a list, it's stored more efficiently, a list takes up that many more bits of memory, so I will just have a structure which is, um, call it mmw, / min / max / window, and then I have got my window which is going to be a list of 6 elements,
and the max is the third argument, / so the mmw sort of structure is just
completely arbitrary, but is just that more efficient than having it as a
list, or something like that, so, it is also more easier to pick off if I treat
that as an object anywhere else in the program, I don’t have to worry
about someone misinterpreting it as a list or something like that as well,
so it means that I’m actually tagging things a bit more, safely.

Right, so we’ve got TimeTotal / and then the new load window is
going to be, um, it is going to have a nice huge long and [mumble]

So, we’re going to have a new load window of mmw / with just
shifting, shifting the old number along, / and the new time total going
on the end, because it is a fixed list, fixed window and there’s not too
many of them, you know, just shifting them along is not to painful to
do with these arguments, you could generalise it so that you take of the
head and scan and put something on the tail and shift as you go down,
building a new list but that would be fairly horrendous in terms of
efficiency.

Perhaps this isn’t quite so bad, not quite so bad.

TimeTotal / and then I’ve got the NewMin / and potentially a NewMax,
and that’s the end of that.

New window structure out.

Right, so what do I want to do, um, so I’ve got new window, now
I’ve got to think about what happens in this initial case, when I’ve got
nothing there.

What I’m going to do okay, what I’m going to do basically I’m going
to write something that is, um, calculate the, um, max, get the sum of
the window, this is where I suppose, um, I do list processing for that.

Right, what I might do is actually, I’ve done this explicitly here, I
might, I might make.

I’m just taking stuff I plonked in the header before, and I’m just going
to pick out the window, the list of the window as an argument / and
pass it in to something else to do the shift along, because then I can just
pass variables as a list a bit more easily, / and pass the list / in to
get_sum / and do my accumulation down the list, I mean, I don’t, I
know it is a fixed number I could just do a sum, I suppose it’s not to
bad is it?
Again, I do it, I do it in a more, I tend to work a bit more generally than specifically, so I will just treat this as a list, and do the matching at the moment generating a new window explicitly as, I know it is a fixed length list, but in future I could just sort of make it more general to do a sort of popping and that other list.

So I'll just cut that.

So call this Win / and then we're going to have a NewWin which we're going to pass out.

So I crack the, sort of, structure that contains the window, min and the max, but I don't crack it, I don't crack it any further than, I don't crack the window bit yet, / I crack the window when I get a new one out.

So that's just the thing of getting the new window out, / that's just a fact, that's just a shift, nothing at all, okay I could generalise that fact thing into a rule to process an arbitrary length list if the window is decided to be bigger at a later stage.

But this is the cheapest way, the most effective way to do it, in this case.

So, get of the sum of the, how shall I do this?

I do it on the current window or the new window, must do it on the last window, sum of the last window, get the sum of the last window, get_sum and then [mumble]

Um, now I've got the sum, I can see if / the sum is greater than the current Max / then NewMax is the Sum, / otherwise if / the Sum is less than the Min / then the NewMin is the Sum, / otherwise / the new min / and the max just stay the same, NewMin is Min, and NewMax is Max.

It's horrible when you can’t indent with tabs.

Is that what I want to do, it looks right.

The only thing I've got to be wary of is, um, when the sum, when the window is invalid, / what I will do is, I think at this stage, I'll probably do the checking as to whether I've got a complete window yet.
The thing is once I know I've got a complete window then I know there after I'm going to have complete windows until the end.

I've also got to cope with the case when I hit, no when I hit the last thing [mumble].

I've just got to cope with when I've got empty things to begin with.

How shall I do that?

I could start the list as a list of nulls in there, so if there's a null in the window then, if there's a null at the start of the window, I only need to look at the front of the window, if there's a null at the front of the window, I haven't got a full window yet.

So, that would be quite easy to test, won't it?

I could start the list as a list of nulls in there, so if there's a null in the window then, if there's a null at the start of the window, I only need to look at the front of the window, if there's a null at the front of the window, I haven't got a full window yet.

So, that would be quite easy to test, won't it?

C Right, what I want to do is somehow basically, so I could do the testing and get sum, because I will be scanning down the list I could, sort of, in effect do the validation of the window at that stage.

So I got make get_sum fail if we haven't got a valid window yet, but then I would have to have an alternative which would allow check_load_window to succeed even when we haven't, you know, which would, just sort of, getting a new window, I haven't put the new window thing in there, have I, brain dead, new_window/Win/, TimeTotal/ NewWin, I don't want to forget that.

Um, I always want to generate a new_window, but then when I, I, sort of, I only want to get the sum of a valid window, so if it an invalid window I only want to, sort of, succeed with the new_window but not have to do any of the testing for, I don't want to test for whether it is less than or greater than, um.

What I could do, what could I do?

It seems a bit gross, see 'cause what I could, what you would normally do I guess is, I need to know to, in effect switch on the Min and Max as soon as I've got a valid window, the first valid window I get is the Min and the Max.

So, what I could is have Min and Max as something which is not a valid number, and I could test on that.

As soon as I get a valid window then I, um, then I substitute that in and off I go, let me think.
Appendices

<p>| A4 | Get_sum window, / I'm going to have to test. | 260 |
| F1 | | 261 |
| C | | 262 |
| I | Um, let me think. | 263 |
| C | If I go down here, if I do something with sum so that it is a number or it's something else, if it's something else then I know I haven't got a valid window. | 264 |
| [pause] | | |
| C | The sum of null then I don't do it. | 265 |
| D1 | Then test becomes [?] the first time, it is isn't it? | 266 |
| G3 | What do I want to do, I want to something so as soon as I get a valid element at the front of the window / then I turn on the Min / and the Max with that sum. | 267 |
| B1 | So what could I do, I could. | 267a |
| D1 | I want to do this less then test if Min and Max, I don't, you know, what I could is start of Min as a big number and Max as 0 so that as soon as I get a Sum that is a valid number then the conditions succeed, but then, you know, what big number do I choose, there is always the situation where one can never guarantee big big number, apart from get maximum, there's probably a predicate that returns maximum int or something in LPA, there's predicates for just about everything. | 268 |
| C | But I really want to, sort of, not have valid Min and Max, I don't want to have a number there until I hit it, I think. | 269 |
| A1 | Nightmare, perhaps if get_sum / returns / null / if there's a null, you know, if it's a null at the front of the list, / otherwise / it will, it will accumulate it. | 270 |
| F1 | | 270a |
| C | | 270b |
| C | Then I can test if sum is null then carry on. | 271 |
| C | If the sum is null just continue, we don't need to, you know, Min and Max can be unassigned, it doesn't matter. | 271a |
| B1 | I've got the case when it first comes in, I don't want to test that every time. | |</p>
<table>
<thead>
<tr>
<th>D1</th>
<th>How shall I do this, I'm just, I'm thinking, you know, that there must be some nice way of doing this, basically, I mean I could do it all, sort of, painfully and, sort of, then there will be several tests every, every time I check the load window, that are really to do the initialisation of the window which is nasty, there must be a nice way, you know, once I realise I've got a full window, I don't need to worry about it, I don't need to worry to much about it there after.</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>What I could do, so what I could do is, is start of with Min / and Max as unsubstantiated variables, they are just unknowns / and have get_sum / and if get_sum hits, if the window is incomplete, um, no I can’t do that, can I.</td>
</tr>
<tr>
<td>D1</td>
<td>As soon as it is completed / then I try to unify, but that not going to work is it, because it is going to be bumped up all the time.</td>
</tr>
<tr>
<td>A1</td>
<td>There must be a nice way of doing this.</td>
</tr>
<tr>
<td>A1</td>
<td>There must be a nice way of doing this.</td>
</tr>
<tr>
<td>[pause]</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Let’s write get_sum while I think about it, I’m going to have to do that any way.</td>
</tr>
<tr>
<td>C</td>
<td>So, I’m going to accumulate a list, lets do it that way for the moment, / although I could do that in new_window, I could it in new_window so why should I do it here?</td>
</tr>
<tr>
<td>D1</td>
<td>It could always be written as a, sort of, list recursive thing if I want to, but at the moment I can do it with one arithmetic computation on 6.</td>
</tr>
<tr>
<td>F1</td>
<td>Get_sum of that is Sum and Sum is A plus B plus C, D plus E.</td>
</tr>
<tr>
<td>D1</td>
<td>I haven’t, you know, by doing it this way, I haven’t precluded, you could rewrite get_sum so that it is recursive and calculated the sum, but I’ve done it this way just because who cares for the moment, it could be done later if it is needed, it won’t break anything else, but here it is more efficient to do it as one ‘is’ calculation rather than recursing and do six ‘is’, calls to is.</td>
</tr>
<tr>
<td>E1</td>
<td>So that will get the sum.</td>
</tr>
<tr>
<td>C</td>
<td>Now I want to, I also want to check that A is valid, don’t I?</td>
</tr>
<tr>
<td>F1</td>
<td>So if A is identical to null, I’m going to start of with null, I’m always going to have a list, [mumble] I have only got 6.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>92</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>D1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>I</td>
</tr>
<tr>
<td></td>
<td>E2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>E1</td>
</tr>
<tr>
<td></td>
<td>A3</td>
</tr>
<tr>
<td></td>
<td>G3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H</td>
</tr>
<tr>
<td>F1</td>
<td>Hang on, test if / the Sum is null / then don’t do anything.</td>
</tr>
<tr>
<td>----</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>C</td>
<td>[pause]</td>
</tr>
<tr>
<td>C</td>
<td>So I could make a big, sort of, alternative test, you know, for alternatives here.</td>
</tr>
<tr>
<td>A3</td>
<td>But I have to make sure of Max and Min bits, is something sensible at some point.</td>
</tr>
<tr>
<td>F1</td>
<td>NewMin is Min / and NewMax is Max.</td>
</tr>
<tr>
<td></td>
<td>[pause]</td>
</tr>
<tr>
<td>D1</td>
<td>No, perhaps there isn’t a pleasant way of doing this, lets see, just do it.</td>
</tr>
<tr>
<td>C</td>
<td>So if Sum is null then just, now, abiding our time, it doesn’t matter what Min and Max are, NewMin, NewMax remain unsubstantiated, for the moment.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, you know, what I could do is, is, sort of.</td>
</tr>
<tr>
<td></td>
<td>[pause]</td>
</tr>
<tr>
<td>C</td>
<td>So what I’ve got to do in this test is to decide whether a Max is known to actually bother to do a comparison.</td>
</tr>
<tr>
<td>I</td>
<td>Because.</td>
</tr>
<tr>
<td></td>
<td>[pause]</td>
</tr>
<tr>
<td>C</td>
<td>So what I could do, this is probably fairly gross, I think, I could do, you know, test whether Max can be unified with Sum, / then NewMax is Sum, because that will fail if, if Max is already there.</td>
</tr>
<tr>
<td></td>
<td>It is just the case where I know that the Sum is not null, so I now know that Sum is something meaningful, so I need to say, I need almost to check have I already got a Max to work with, and one way of doing that is trying to unify Max with my current Sum, and if you can that then Max is unsubstantiated or Max is equivalent in which case NewMax just remains the same anyway, or becomes unified, and.</td>
</tr>
<tr>
<td>C</td>
<td>Then I’ve got to cope with when I want to do the Min as well, oh urk, urk, urk.</td>
</tr>
<tr>
<td>D1</td>
<td>I’m missing something very obvious here, please.</td>
</tr>
<tr>
<td>C</td>
<td>Perhaps I should just fail / if it is an invalid sum, / I’ve still got the problem of when to turn on.</td>
</tr>
<tr>
<td>D1</td>
<td>[pause]</td>
</tr>
</tbody>
</table>

- 226 -
So the first time I get a valid window, / I want to make Max / and Min equivalent to the sum of that window, / because then I don’t have to worry about picking a little number and picking a high number, you know, that is the first Min and Max.

So, I’m always going to have this problem whenever I’m, whenever I do it, I’m never going to know whether I’ve, I’ve currently got a Max or a Min yet or not.

Unless I’ve some other of global flag that says yes you’ve got a full window so far, I suppose, I suppose what I could do is do a test on passing the time, hang on, no, I could pass in the current time, and if the current time is above the window size, hang on, no, then we’ve got to worry about the future if the window size is changed that the test for that is changed as well.

No lets just do it this way, lets just get it working.

The Sum’s null / we don’t do anything.

Otherwise if it is not null then, um.

So if / Max and Min can be unified with Sum, so I’m going to do two unification tests every cycle which once you get a full window are going to be wasted, but I can’t see a nice way of doing this test.

That’s sort of reasonably robust, NewMin is Sum.

So comment that, that’s the case when first full window.

Were they can share by shear fluke but that shouldn’t make any difference, ‘cause there’s no change to the Min and Max if that does happen, so I’m not losing anything if I do hit that condition that later on, anyway.

Otherwise if / the Sum is greater than Max / then NewMax is Sum.

If / the Sum is less / NewMin is Sum; / and NewMin is just Min / and NewMax is just Max.

Otherwise / NewMin is Min / and NewMax is Max when none of these others hold, does that make sense?
Hang on, yes because that could be when Max isn’t the same as, the Sum isn’t the same as Max and Sum isn’t the Min, so I think that should work.

It’s a bit tacky, but I think we’re going to have to do something tacky anyway, something a bit cumbersome to cope with, you know, waiting until you’ve got a full window.

I hope there is some nice ideal solution to this, so I can go home not worrying about it, to solve this nicely.

So, is that, I’ve done that, getting the Min, Max, the thing of the window now, haven’t I?

So I’ve just got to do the reporting at the end, so what happens is at the end when I do the day totals, report_day, I do DayTotal, and I pass in the LoadWindow as well now, and then I do that report day.

The LoadWindow is the mmw, the Max number, I don’t need details about the actual window itself any more, I just need the Max and Min, did call them Max and Min, probably did, Min Max there we go.

Um, and so on the end here / I want to say write the busiest 60 minutes, oh, started at, oh bollocks, I’ve should’ve read this shouldn’t I.

‘Always read the exam paper before you start writing.’

So I know what the Max and the Min are but I don’t know when it bloody started, though.

Um, so I don’t need to keep the Min and the Max actually, oh, I do, I need to keep both don’t I, oh bollocks.

Oh, sorry, beep.

So I’m going to have, um, the MaxTime and the MinTime.

Here we go this is why I was glad I did this as one package because its means I just need to change one [background noise] otherwise it is just past as a single unit.

So started, ‘I’ve started so I’ll finished’, started at, um, I should’ve read this first, shouldn’t I, I should’ve looked at what I was doing, prat, perhaps I could generalise write_hour, so that when it takes minutes it, um, does print out the minute part of it as well.

Where I call it up there it is always an hour.

So the minutes are always going to be zero, so I could do, perhaps I’ll use write_hour, I’ve still got the logic for the rest of it anyway.
<p>| A2 | So if I just maintain the minutes MaxTime / and MinTime in minutes, / and I'll just use the write_hour, modify write_hour to actually print it out. |
| F1 | Started at, and do, change it to write_time, now, write_hour is MaxTime. |
| A4 | The lightest. |
| F1 | FI Started at, and do, change it to write_time, now, write_hour is MaxTime. |
| 107 A3 | Right so I've got to change write_hour to write_time, / now, Mins, so now I've got to cope with write_hour. |
| F1 | Pass in TotalMins, / and work out the Hour / and then get the minute parts out, that's the rem, is TotalMin mod 60, / and write UseHour, / write full spot, is it justified as well, the zeros, I suppose I better do it that way. |
| H | This is when, that's when a formatted write out, to say put out zeros padded with zeros to the width two. |
| I | [looking at format write - fw] |
| D1 | Lets not worry about it lets just do this test. |
| 108 F1 | Write dot / and then if / the Mins, Mins is less than 10 / then / write 0, I'm not to worried about writing out stuff [mumble]. |
| F1 | Otherwise / true, / and then write the Mins. |
| G1c | [talking about time constraint] |
| F1 | Well I have every confidence in this nearly working, I just need to sort out when I hit a NewMin, NewMax, that I also put in NewMinTime / and NewMaxTime. |
| A3 | So here I just need to do some initialisation, oh yuck. |
| H | Thousands of variables, um I have got to do NewMaxTime is, lets break this down. |</p>
<table>
<thead>
<tr>
<th>E1</th>
<th>NewMaxTime is, max is, we've got a new time of, I'm not keeping the current time am I?</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2</td>
<td>Hang on, I know what the current time is, no I don't, TimeTotal, I could pass in the current time, hang on, how do we know, minutes it started at, um.</td>
</tr>
<tr>
<td>C</td>
<td>What do I do, current time minus 60.</td>
</tr>
<tr>
<td>B2</td>
<td>I have to pass in the time, this is getting gross now, oh dear.</td>
</tr>
<tr>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So NewMaxTime is current Time / and NewMinTime is just as it was, NewMinTime is MinTime, okay.</td>
</tr>
<tr>
<td>F1</td>
<td>What have we, NewMinTime is Time / and NewMaxTime is just as it was before.</td>
</tr>
<tr>
<td>E1</td>
<td>Otherwise, no hang on, I'm doing all this completely up the shute, ain't I, I'm not reading my own code.</td>
</tr>
<tr>
<td>C</td>
<td>That should be [mumble].</td>
</tr>
<tr>
<td>E1</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>NewMaxTime is MaxTime / and NewMinTime is MinTime.</td>
</tr>
<tr>
<td>B2</td>
<td>I've got to make sure I'm pass in the current time, in check time, in check_load_window, time taken I want to pass, um, NewTime, what do I want to pass in, NewTime, because I'm doing it for that.</td>
</tr>
<tr>
<td>114 A3</td>
<td>And then report it at the end, I have to do that adjustment so that I'm doing it for the new time I've just read in, so the newest time would be on the new tail there, so when that window started is that time minus 60, isn't it?</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>So I could just.</td>
</tr>
<tr>
<td></td>
<td>[pause]</td>
</tr>
<tr>
<td>F1</td>
<td>I've got the Min and MaxTime, I just, I just do, StartMaxTime is MaxTime minus 60.</td>
</tr>
<tr>
<td>F1</td>
<td>StartMinTime.</td>
</tr>
<tr>
<td>H</td>
<td>So that's when I start debugging, now.</td>
</tr>
</tbody>
</table>
I.2. Prolog Solution of A4

The code below is the final solution produced by A4. This code is used to create the initial goal hierarchy.

```
go:-
    read_in_datafile.

read_in_datafile:-
    open(survey),
    read_in_datafile(0, 0, 0, [], 0),
    close(survey).

read_in_datafile(Timestamp, Total, HourTotal, LoadWindow, DayTotal):-
    get(Ch),
    ( Ch == 0'1 ->
        NewTimestamp is Timestamp + 10,
        LastHourTotal is HourTotal + Total,
        check_report_hour(NewTimestamp, LastHourTotal, NewHourTotal, DayTotal,
            NewDayTotal),
        check_load_window(LoadWindow, Total, NewTimestamp, NewLoadWindow),
        read_in_datafile(NewTimestamp, 0, NewHourTotal, NewLoadWindow, NewDayTotal)
    ; Ch == 0'2 ->
        NewTimestampTotal is Total + 1
    ).
```
read_in_datafile(TimeStamp, NewTimeTotal, HourTotal, LoadWindow, DayTotal)

; Ch == -1 ->

report_day(DayTotal, LoadWindow)

; write('Unrecognised input: '), put(Ch),

read_in_datafile(TimeStamp, TimeTotal, HourTotal, LoadWindow, DayTotal)

).

check_report_hour(NewTime, LastHourTotal, NewHourTotal, DayTotal, NewDayTotal):-

( NewTime mod 60 == 0 ->

write_hour(NewTime),

write(' : '),

write(HourTotal), nl,

NewHourTotal = 0,

NewDayTotal is DayTotal + LastHourTotal

; NewHourTotal = LastHourTotal,

NewDayTotal = DayTotal

).

check_load_window(mmww(Win, Min, Max, MinTime, MaxTime), TimeTotal, Time,

mmww(NewWin, NewMin, NewMax, NewMinTime,

NewMaxTime)):-

new_window(Win, TimeTotal, NewWin),

- 232 -
new_window([_, A, B, C, D, E], Time, [A, B, C, D, E, TimeTotal]).

get_sum([A, B, C, D, E, F], Sum):-
   !, % red replaces test for 6 element below
   Sum is A + B + C + D + E.
get_sum(\, ').
write_time(TotalMins):-
    Hour is TotalMins // 60 - 1,
    /% we've already hit next hour
    ( Hour > 12 ->
        UseHour is Hour - 12,
        AMPM = pm
    ; UseHour = Hour,
        AMPM = am
    ),
    Mins is TotalMin mod 60,
    write(UseHour),
    write('.'),
    ( Mins < 10 -> write('0')
    ; true
    ),
    write(Mins),
    write(AMPM).

report_day(DayTotal, mmw(_,_,_ , MaxTime, MinTime)):-
    nl,
    write('Total number of cars: '),
    write(DayTotal), nl, nl,
    Average is DayTotal / 24,
    write('The average number of cars per hour: '),
I.3. Goal Hierarchy of A4

The following pages contain a pictorial representation of A4’s goal hierarchy. Each node contains (1) the node’s address, (2) a short description of the goal, and (3) the numbers of the actions performed on the node. The goal hierarchy is derived from the program produced by A4, the verbal reports, and the program at various stages of completion.

write(Average), nl, nl,
write('The busiest 60 minutes started at: '),
StartMaxTime is MaxTime - 60,
write_time(StartMaxTime), nl,
write('The lightest 60 minutes started at: '),
StartMinTime is MinTime - 60,
write_time(StartMinTime), nl.
I.9.3.4.6.2.1.X
Write Hour
[101, 112, 330, 331c 338]

X.1
Input NewTime
[102, 113, 340]

X.2
Amend Time is Starting Time
[103, 114a]

X.3
Format NewTime
[114]

X.4
Write Formatted Time
[117]

X.3.1
Calculate Hour
[115, 127a, 341]

X.3.2
Reduce Hour by One
[116]

X.3.3
Calculate if am
[118, 119a]

X.3.4
Calculate Mins
[339, 342]

X.4.1
Write UseHour
[122, 343]

X.4.2
Write ‘.’
[344, 346]

X.4.3
Write ‘.00’
[123, 126, 344a]

X.4.4
Justified with zeros
[345]

X.4.5
Write Mins
[331, 349]

X.4.6
Write AMPM
[127]

X.3.3.1
Test
[119, 119b]

X.3.3.2
Yes
[119c]

X.3.3.3
No
[120a]

X.3.3.2.1
UseHour is Hour-12
[120]

X.3.3.2.2
AMPM
[124]

X.3.3.3.1
UseHour is 'pm'
[121]

X.3.3.3.2
UseHour is 'am'
[125]

X.4.4.1
Test Mins<10
[347]

X.4.4.2
Yes
[347a]

X.4.4.3
No
[348a]

X.4.4.2.1
Write '0'
[348]

X.4.4.3.1
True
[348b]
Get Sum of Window

1.1.9.3.7.5.5

Input Window

1.1.9.3.7.5.5.1
Cut
[284, 285a]

1.1.9.3.7.5.5.2
Test
[261]

1.1.9.3.7.5.5.3
First Condition
[270b, 288, 283, 284b]

1.1.9.3.7.5.5.4
Second Condition
[251b, 285, 290]

1.1.9.3.7.5.5.5
Return Sum
[269, 287]

1.1.9.3.7.5.5.5.1
Cut
[284, 285a]

1.1.9.3.7.5.5.5.2
Sum is sum of elements
[234a, 270c, 276, 278, 289]

1.1.9.3.7.5.5.5.3
Fail
[253a]

1.1.9.3.7.5.5.5.4.1
Sum is Null
[263, 270, 286, 291]

1.1.9.3.7.5.5.5.4.2
Succeed
[259b]
Appendices

1.4. Order In Which A4 Constructed His Goal Hierarchy

The Table below lists all the actions that construct parts of the goal hierarchy shown in the diagrams above. In the table, the transition from the previous node to the next node is determined as to whether it conforms to the breadth-first, the children-first and the depth-first global control strategies (for the table below, the conservative definition of the control strategies was used). At the end of the table is a count of the number of transitions that conform and the number of transitions that diverged from each control strategy. The percentage of conformance to each control strategy is also provided. A similar table is created using the liberal definition of control strategies. Differences between the two tables are reviewed to resolve the differences.

Table 1.2: Comparing Transition against Rules of Structured Control Strategies

<table>
<thead>
<tr>
<th>Action Number</th>
<th>Address of Node Traversing To</th>
<th>Does Transition Conform to the Global Control Strategy?</th>
<th>Breadth-First</th>
<th>Children-First</th>
<th>Depth-First</th>
<th>Breadth or Children-First</th>
<th>Breadth or Depth-First</th>
<th>Children or Depth-First</th>
<th>Top-Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>6</td>
<td>1.1</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>7</td>
<td>1.2</td>
<td>√</td>
<td></td>
<td>x</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>8</td>
<td>1.3</td>
<td>√</td>
<td></td>
<td>x</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>11</td>
<td>1.1</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>12</td>
<td>1.1.1</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>20a</td>
<td>1.1.9</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>21</td>
<td>1.1.2</td>
<td>√</td>
<td></td>
<td>x</td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>24</td>
<td>1.1.10</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>25</td>
<td>1.1.9</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>26</td>
<td>1.1.9.3.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1.1.9.3.6</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>27a</td>
<td>1.1.9.3.7</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>28</td>
<td>1.1.9.3.7.1</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1.1.9.3.7.2</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29a</td>
<td>1.1.9.3.8</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1.1.9.3.8.1</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1.1.9.3.8.3</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1.9.3.9</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----------------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1.1.9.3.9.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>33a</td>
<td>1.1.9.3.9.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1.1.9.3.7.7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1.1.9.3.8.5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1.1.9.3.10</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1.1.9.3.10.2</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>44a</td>
<td>1.1.9.3.10.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1.1.9.3.10.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>1.1.9.3.7.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1.1.9.3.3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>1.1.9.3.5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>66b</td>
<td>1.1.9.3.7.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>1.1.9.3.7.4.3</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>67a</td>
<td>1.1.9.3.7.4.1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>1.1.9.3.9.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>1.1.9.3.8.3</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>1.1.9.3.8.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>75b</td>
<td>1.1.9.3.10.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1.1.9.3.7.4.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>1.1.9.3.7.4.8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>1.1.9.3.7.7</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>1.1.9.3.7.4.7</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1.1.9.3.7.4.6.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>103a</td>
<td>1.1.9.3.7.4.6.2.1.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>1.1.9.3.7.4.6.2.1.2</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>1.1.9.3.7.4.6.2.1.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>1.1.9.3.7.4.6.2.3</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>1.1.9.3.7.4.6.2.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>1.1.9.3.7.4.6.3</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>1.1.9.3.7.4.6.3.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1.1.9.3.7.4.6.3.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>1.1.9.3.7.4.6.2.1.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>1.1.9.3.7.4.6.2.1.1.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>1.1.9.3.7.4.6.2.1.1.3.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1.9.3.7.4.6.2.1.1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>119a</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>119b</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>119c</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.2.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120a</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.3</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.3.1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>1.1.9.3.7.4.6.2.1.1.4.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>1.1.9.3.7.4.6.2.1.1.4.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.2.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>1.1.9.3.7.4.6.2.1.1.3.3.3.2</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>1.1.9.3.7.4.6.2.1.1.4.5</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1.1.9.3.9.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>1.1.9.3.9.2.1</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>1.1.9.3.7.4.6.2.1.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130a</td>
<td>1.1.9.3.9.2.3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>1.1.9.3.9.2.4.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>1.1.9.3.9.2.4.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>1.1.9.3.9.2.4.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>1.1.9.3.9.2.5.1</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>1.1.9.3.9.2.5.2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>1.1.9.3.9.2.5.3</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>1.1.9.3.9.2.5.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>1.1.3</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>1.1.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>1.1.9.3.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>1.1.6</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>166a</td>
<td>1.1.9.3.8.5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>1.1.9.3.7.6</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>1.1.9.3.8.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>1.1.9.3.8.4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>1.1.9.3.7.5</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>174a</td>
<td>1.1.9.3.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>1.1.9.3.7.5</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>181</td>
<td>1.1.9.3.7.5.1</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>182</td>
<td>1.1.9.3.7.5.2</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>183</td>
<td>1.1.9.3.7.5.12</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184</td>
<td>1.1.9.3.7.7</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184a</td>
<td>1.1.9.3.4</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184b</td>
<td>1.1.9.3.8.5</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184c</td>
<td>1.1.9.3.10.4</td>
<td>✓ ✓ ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>1.1.9.3.7.4.2</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>1.1.9.3.7.4.2</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>1.1.9.3.7.4.5</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>1.1.9.3.7.4.6.2.2</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>198a</td>
<td>1.1.9.3.7.4.6.3.1</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>1.1.9.3.7.4.2</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>1.1.9.3.7.4.3</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>1.1.9.3.7.4.2</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>1.1.9.3.7.3</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>211</td>
<td>1.2</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>1.3</td>
<td>✓ ✓ x ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>1.1.5</td>
<td>x ✓ x ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>1.1.9.3.7.5</td>
<td>x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215a</td>
<td>1.1.9.3.7.5.1</td>
<td>x ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215b</td>
<td>1.1.9.3.7.5.2</td>
<td>✓ ✓ x ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215c</td>
<td>1.1.9.3.7.5.12</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>1.1.9.3.7.5.1.2</td>
<td>x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>221</td>
<td>1.1.9.3.7.5.1.3</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>1.1.9.3.7.5.1.1</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>227</td>
<td>1.1.9.3.7.5.12.1</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>1.1.9.3.7.5.12.2</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>229</td>
<td>1.1.9.3.7.5.12.3</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>234</td>
<td>1.1.9.3.7.5.5</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>235</td>
<td>1.1.9.3.7.5.1.1</td>
<td>x x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>236</td>
<td>1.1.9.3.7.5.12.1</td>
<td>x x x x x x x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>239</td>
<td>1.1.9.3.7.5.4</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>241</td>
<td>1.1.9.3.7.5.5</td>
<td>✓ ✓ x ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>1.1.9.3.7.5.8</td>
<td>✓ ✓ x ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>242a</td>
<td>1.1.9.3.7.5.8.1</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>243</td>
<td>1.1.9.3.7.5.8.2</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>1.1.9.3.7.5.9</td>
<td>x x ✓ x ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>244a</td>
<td>1.1.9.3.7.5.9.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>1.1.9.3.7.5.9.2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>1.1.9.3.7.5.11</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>1.1.9.3.7.5.11.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>1.1.9.3.7.5.11.2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>254</td>
<td>1.1.9.3.7.5.4</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>1.1.9.3.7.5.4.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1.1.9.3.7.5.4.2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>257</td>
<td>1.1.9.3.7.5.4.4</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>1.1.9.3.7.5.5</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>261</td>
<td>1.1.9.3.7.5.5.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>269</td>
<td>1.1.9.3.7.5.5.5</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>278</td>
<td>1.1.9.3.7.5.5.3.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>279</td>
<td>1.1.9.3.7.5.5.2</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>285</td>
<td>1.1.9.3.7.5.5.4</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>285a</td>
<td>1.1.9.3.7.5.5.3.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>286</td>
<td>1.1.9.3.7.5.5.4.2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>292a</td>
<td>1.1.9.3.7.5.6</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>293</td>
<td>1.1.9.3.7.5.6.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>304</td>
<td>1.1.9.3.7.5.7</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>305</td>
<td>1.1.9.3.7.5.7.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>305a</td>
<td>1.1.9.3.7.5.7.2</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>306</td>
<td>1.1.9.3.7.5.7.3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>311</td>
<td>1.1.9.3.7.5.8.3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>312</td>
<td>1.1.9.3.7.5.9.3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>324a</td>
<td>1.1.9.3.9.2.6.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>332</td>
<td>1.1.9.3.9.2.6.3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>1.1.9.3.9.2.6.4</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td>1.1.9.3.9.2.7.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>336</td>
<td>1.1.9.3.9.2.7.3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>337</td>
<td>1.1.9.3.9.2.7.4</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>338</td>
<td>1.1.9.3.9.2.7.3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>340</td>
<td>1.1.9.3.9.2.7.3.1</td>
<td>x</td>
<td>x</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>341</td>
<td>1.1.9.3.9.2.7.3.3.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>342</td>
<td>1.1.9.3.9.2.7.3.3.4</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>344a</td>
<td>1.1.9.3.9.2.7.3.4.3</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 246 -
<table>
<thead>
<tr>
<th></th>
<th>1.1.9.3.9.2.7.3.4.4</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>345</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>347</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>347a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>348</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>348a</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>348b</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>350</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>351</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>351a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>351b</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>354</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>355</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>356</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>357</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>359</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>360</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>362</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>363</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Number of Transitions Conforming</th>
<th>86</th>
<th>88</th>
<th>109</th>
<th>91</th>
<th>133</th>
<th>132</th>
<th>135</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Transitions Diverging</td>
<td>94</td>
<td>92</td>
<td>89</td>
<td>89</td>
<td>47</td>
<td>48</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Percentage of Transitions Conforming</td>
<td>48%</td>
<td>49%</td>
<td>61%</td>
<td>51%</td>
<td>74%</td>
<td>73%</td>
<td>75%</td>
</tr>
</tbody>
</table>
J. Individual Results and Statistical Tests

J.1. Comparison of Order of Sub-Problems.

Table J.1: Key to Contents of the Cell's in Table J.2

<table>
<thead>
<tr>
<th>Expect Behaviour</th>
<th>Swap</th>
<th>Kendall's Tau</th>
</tr>
</thead>
<tbody>
<tr>
<td>z Score</td>
<td></td>
<td>Correlation</td>
</tr>
</tbody>
</table>

Using Kendall's τ, the observed orders in which the participants pursue the sub-problems were compared against the expected orders in which the participants pursued the sub-problems.

\[
\tau = 1 - \frac{2(\text{Number of Swaps})}{\text{Number of Pairs of Objects}}
\]

The number of swaps is the number of pairs that have to be switched in order (e.g. observed) to transform it into the other order (e.g. expected). Where there are 4 objects in each order, the number of pairs of objects is 6.

From τ, the z score of statistical significance can be calculated.

\[
z = \frac{\tau}{s_r}, \text{ where } s_r = \sqrt{\frac{2(2N + 5)}{9N(N - 1)}}
\]

As N, the number of objects is 4, then \(s_r = 0.49\). From z, the correlation between the expected and the observed orders can be calculated. For each participant and for each hypothesis, Table J.2. shows (1) the number of swaps, (2) Kendall’s τ, (3) the z score, and (4) the correlation.
## Table J.2: Testing Correlation Between Expected and Observed Order of Sub-Problems using Kendall's $\tau$

<table>
<thead>
<tr>
<th>Participant</th>
<th>Observed</th>
<th>Least Dependent First</th>
<th>Easy First</th>
<th>Difficult First</th>
<th>Analogies and Least Dependent First</th>
<th>Analogies and Easy First</th>
<th>Analogies and Difficult First</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>dbca</td>
<td>abcd</td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>dbca</td>
<td>1 0.67</td>
<td>abdc</td>
<td>dbca</td>
<td>2 0.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1.00</td>
<td>2 0.33</td>
<td>0.68 50%</td>
<td>2.04 96%</td>
<td>1.36 82%</td>
<td>0.00 0%</td>
</tr>
</tbody>
</table>
J.2. Complexity of Solution for Sub-Problems

Table J.3: Testing trend of Complexity increase with Difficulty of Sub-Problem using Page’s L Tread Test for Related Samples

<table>
<thead>
<tr>
<th>u</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
<th>Total L</th>
<th>( \sum Ru )</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>62 (3)</td>
<td>64 (4)</td>
<td>55 (1)</td>
<td>59 (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>28 (1)</td>
<td>153 (4)</td>
<td>90 (3)</td>
<td>79 (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>5 (1)</td>
<td>81 (3)</td>
<td>80 (2)</td>
<td>108 (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>77 (2)</td>
<td>43 (1)</td>
<td>124 (3)</td>
<td>127 (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>15 (1)</td>
<td>90 (3)</td>
<td>79 (2)</td>
<td>104 (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>21 (1)</td>
<td>93 (2)</td>
<td>171 (4)</td>
<td>159 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>9 (1)</td>
<td>72 (4)</td>
<td>60 (3)</td>
<td>49 (2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \bar{X} )</td>
<td>31.00</td>
<td>85.14</td>
<td>94.14</td>
<td>97.86</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>25.64</td>
<td>31.85</td>
<td>37.69</td>
<td>35.82</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>10</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ru</td>
<td>10</td>
<td>42</td>
<td>54</td>
<td>84</td>
<td>190</td>
<td></td>
</tr>
</tbody>
</table>

Participant C5 is excluded from the table as he did not rank the sub-problems in order of difficulty. With 7 participants and 4 conditions, the critical value of L at the 95% confidence level is 189. L is 190. There is a significant trend as L is greater than L critical. The complexity of a solution increases with the difficulty of the problem.
## J.3. Time Taken to Solve Sub-Problems

### Table J.4: Testing trend of Time increase with Difficulty of Sub-Problem using Page's L Tread Test for Related Samples

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
<th>Total</th>
<th>( \sum Ru )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>8.37 min</td>
<td>12.95 min</td>
<td>10.84 min</td>
<td>31.92 min</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>12.84 min</td>
<td>16.26 min</td>
<td>12.03 min</td>
<td>10.43 min</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>8.25 min</td>
<td>18.30 min</td>
<td>10.35 min</td>
<td>15.07 min</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>9.67 min</td>
<td>12.61 min</td>
<td>17.81 min</td>
<td>21.96 min</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>8.57 min</td>
<td>9.30 min</td>
<td>24.00 min</td>
<td>21.84 min</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>4.00 min</td>
<td>7.50 min</td>
<td>27.47 min</td>
<td>21.07 min</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>9.05 min</td>
<td>10.95 min</td>
<td>17.05 min</td>
<td>14.00 min</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>8.68 min</td>
<td>12.55 min</td>
<td>17.08 min</td>
<td>19.47 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>2.41</td>
<td>3.50</td>
<td>6.17</td>
<td>6.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>9</td>
<td>19</td>
<td>21</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ru</td>
<td>9</td>
<td>38</td>
<td>63</td>
<td>84</td>
<td>194</td>
<td></td>
</tr>
</tbody>
</table>

Participant C5 is excluded from the table as he did not rank the sub-problems in order of difficulty. With 7 participants and 4 conditions, the critical value of \( L \) at the 95% confidence level is 189. \( L \) is 194. There is a significant trend as \( L \) is greater than \( L \) critical. The time spent producing a solution increases with the difficulty of the problem.
### J.4. Percentage of Transitions Devoted to Each Sub-Problem

Table J.5: Testing trend of Transition increase with Difficulty of Sub-Problem using Page’s L Trend Test for Related Samples

<table>
<thead>
<tr>
<th>u</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
<th>Total</th>
<th>[ \sum R_u ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>13.74%</td>
<td>22.84%</td>
<td>16.98%</td>
<td>46.44%</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>22.25%</td>
<td>33.07%</td>
<td>25.29%</td>
<td>19.40%</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>10.67%</td>
<td>9.92%</td>
<td>26.12%</td>
<td>34.85%</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>16.77%</td>
<td>24.53%</td>
<td>34.21%</td>
<td>24.49%</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>13.34%</td>
<td>18.76%</td>
<td>36.01%</td>
<td>31.89%</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>6.30%</td>
<td>10.95%</td>
<td>43.34%</td>
<td>39.42%</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>18.57%</td>
<td>24.85%</td>
<td>35.24%</td>
<td>21.34%</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>[ \bar{X} ]</td>
<td>14.52%</td>
<td>20.70%</td>
<td>31.03%</td>
<td>31.12%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>4.86</td>
<td>7.60</td>
<td>8.09</td>
<td>9.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8</td>
<td>18</td>
<td>24</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ru</td>
<td>8</td>
<td>36</td>
<td>72</td>
<td>80</td>
<td>196</td>
<td></td>
</tr>
</tbody>
</table>

Participant C5 is excluded from the table as he did not rank the sub-problems in order of difficulty. With 7 participants and 4 conditions, the critical value of \( L \) at the 95% confidence level is 189. \( L \) is 196. There is a significant trend as \( L \) is greater than \( L \) critical. The percentage of transition devoted to a solution increases with the difficulty of the problem.
### J.5. Time Taken to Complete Solution By Order of Being Solved

Table J.6: Testing trend of Transition increase with Difficulty of Sub-Problem using Page’s L Tread Test for Related Samples

<table>
<thead>
<tr>
<th></th>
<th>First 4.0</th>
<th>Second 1.0</th>
<th>Third 2.5</th>
<th>Fourth 2.5</th>
<th>Total L</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>+48.78% (4.0)</td>
<td>-44.75% (1.0)</td>
<td>+24.36% (2.5)</td>
<td>-33.56% (2.5)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>+18.68% (2.5)</td>
<td>-5.81% (2.5)</td>
<td>-33.94% (1.0)</td>
<td>+53.25% (4.0)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>-2.31% (2.5)</td>
<td>+79.11% (4.0)</td>
<td>-5.30% (2.5)</td>
<td>-40.52% (1.0)</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>+25.06% (4.0)</td>
<td>-20.73% (1.0)</td>
<td>-14.27% (2.5)</td>
<td>+15.58% (2.5)</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>+25.77% (2.5)</td>
<td>-36.15% (1.0)</td>
<td>+39.80% (4.0)</td>
<td>-29.42% (2.5)</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>+23.03% (4.0)</td>
<td>+2.39% (2.5)</td>
<td>-25.75% (1.0)</td>
<td>-17.22% (2.5)</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td>-36.46% (2.5)</td>
<td>-59.00% (1.0)</td>
<td>+14.61% (2.5)</td>
<td>+36.66% (4.0)</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td>-18.09% (1.0)</td>
<td>+9.10% (4.0)</td>
<td>+9.07% (2.5)</td>
<td>+9.09% (2.5)</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>+10.56%</td>
<td>-9.48%</td>
<td>+1.07%</td>
<td>-0.77%</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>25.79</td>
<td>40.13</td>
<td>23.74</td>
<td>32.46</td>
<td>209</td>
</tr>
<tr>
<td>R</td>
<td>23</td>
<td>17</td>
<td>19</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Ru</td>
<td>92</td>
<td>17</td>
<td>46</td>
<td>54</td>
<td></td>
</tr>
</tbody>
</table>

The second solution tackled was ranked first as being the quickest to complete. The third and fourth were equally ranked as being the next quickest to complete. The first solution tackled was ranked last as being the slowest to complete. With 8 participants and 4 conditions, the critical value of L at the 95% confidence level is 214. L is 209. There is no significant trend as L is less than L critical. The time taken to complete a solution is not dependent on the order in which the solutions are tackled.
J.6. **Mean Time Spent Deliberating at Start of a Procedure**

**Table J.7: Independent T Test - difference between ‘Mean Deliberation at Start of a Procedure’ for Strategies Used**

<table>
<thead>
<tr>
<th>Participants</th>
<th>Breadth Not Breadth</th>
<th>Strategy Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4</td>
<td>88.44</td>
<td>52.12</td>
</tr>
<tr>
<td>D6</td>
<td>83.84</td>
<td>79.61</td>
</tr>
<tr>
<td>D8</td>
<td>70.52</td>
<td>37.18</td>
</tr>
<tr>
<td>D12</td>
<td>65.28</td>
<td>51.88</td>
</tr>
<tr>
<td>D9</td>
<td>37.16</td>
<td>34.91</td>
</tr>
<tr>
<td>D2</td>
<td>61.41</td>
<td>34.96</td>
</tr>
<tr>
<td>D3</td>
<td>58.78</td>
<td>49.78</td>
</tr>
<tr>
<td>D10</td>
<td>34.96</td>
<td>14.43</td>
</tr>
<tr>
<td>D11</td>
<td></td>
<td>1.80</td>
</tr>
</tbody>
</table>

With 13 participants, the critical value of t at the 95% confidence level for a one-tail test is 1.80. T is 3.18. There is a significant difference between the two populations as t is greater than t critical. Programmers using the breadth-first control strategy spend more time deliberating at the start of a procedure than programmers using alternative control strategies.
### Mean Time Spent between each Keystroke

**Table J.8: Independent T Test - difference between 'Mean Time between Keystrokes' for Strategies Used**

<table>
<thead>
<tr>
<th>Strategy Used</th>
<th>Participants</th>
<th>Mean Time between Keystrokes (sec.)</th>
<th>Participants</th>
<th>Mean Time between Keystrokes (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth</td>
<td>D4</td>
<td>2.00</td>
<td>D1</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>D6</td>
<td>1.99</td>
<td>D5</td>
<td>2.20</td>
</tr>
<tr>
<td></td>
<td>D8</td>
<td>1.05</td>
<td>D7</td>
<td>2.88</td>
</tr>
<tr>
<td></td>
<td>D12</td>
<td>0.74</td>
<td>D9</td>
<td></td>
</tr>
<tr>
<td>Not Breadth</td>
<td>D4</td>
<td></td>
<td>D11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D6</td>
<td></td>
<td>D13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D8</td>
<td></td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D12</td>
<td></td>
<td>D3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>D4</td>
<td></td>
<td>D10</td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>1.45</td>
<td></td>
<td>Mean</td>
<td>1.63</td>
</tr>
<tr>
<td>Variance</td>
<td>0.56</td>
<td></td>
<td>Variance</td>
<td>0.64</td>
</tr>
<tr>
<td>t</td>
<td>0.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t crit</td>
<td>1.80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With 13 participants, the critical value of $t$ at the 95% confidence level for a one-tail test is 1.80. $T$ is 0.45. There is no significant difference between the two populations as $t$ is less than $t$ critical. Programmers using the breadth-first control strategy spend the same amount of time between keystrokes than programmers using alternative control strategies.
### J.8. Number of Errors in a Solution

Table J.9: Independent T Test - difference between 'Number of Errors in Solution' for Strategies Used

<table>
<thead>
<tr>
<th>Participants</th>
<th>Number of Errors in Solution</th>
<th>Strategy Used</th>
<th>Number of Errors in Solution</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4</td>
<td>3.00</td>
<td>Breadth</td>
<td>7.00</td>
<td>D1</td>
</tr>
<tr>
<td>D6</td>
<td>4.00</td>
<td></td>
<td>5.00</td>
<td>D5</td>
</tr>
<tr>
<td>D8</td>
<td>4.00</td>
<td></td>
<td>5.00</td>
<td>D7</td>
</tr>
<tr>
<td>D12</td>
<td>3.00</td>
<td></td>
<td>3.00</td>
<td>D9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
<td>D13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7.00</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11.00</td>
<td>D3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9.00</td>
<td>D10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6.00</td>
<td>D11</td>
</tr>
<tr>
<td>Mean</td>
<td>3.50</td>
<td></td>
<td>6.33</td>
<td>Mean</td>
</tr>
<tr>
<td>Variance</td>
<td>0.50</td>
<td></td>
<td>2.36</td>
<td>Variance</td>
</tr>
<tr>
<td>t</td>
<td>2.19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t crit</td>
<td>1.80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With 13 participants, the critical value of t at the 95% confidence level for a one-tail test is 1.80. T is 2.19. There is a significant difference between the two population as t is greater than t critical. Programmers using the breadth-first control strategy produce fewer errors in a solution than programmers using alternative control strategies.
J.9. Amount of a Solution Completed

Table J.10: Independent T Test - difference between 'Amount of Solution Completed' for Strategies Used

<table>
<thead>
<tr>
<th>Participants</th>
<th>Breadth Strategy Used</th>
<th>Not Breadth Strategy Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amount of Solution Completed (0-5)</td>
<td>Amount of Solution Completed (0-5)</td>
</tr>
<tr>
<td>D4</td>
<td>3.50</td>
<td>3.00</td>
</tr>
<tr>
<td>D6</td>
<td>4.50</td>
<td>3.50</td>
</tr>
<tr>
<td>D8</td>
<td>5.00</td>
<td>2.00</td>
</tr>
<tr>
<td>D12</td>
<td>5.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>5.00</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>3.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>5.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Mean</td>
<td>4.50</td>
<td>3.39</td>
</tr>
<tr>
<td>Variance</td>
<td>0.61</td>
<td>1.05</td>
</tr>
<tr>
<td>t</td>
<td>1.82</td>
<td></td>
</tr>
<tr>
<td>t crit</td>
<td>1.80</td>
<td></td>
</tr>
</tbody>
</table>

With 13 participants, the critical value of t at the 95% confidence level for a one-tail test is 1.80. T is 1.82. There is a significant difference between the two population as t is greater than t critical. Programmers using the breadth-first control strategy complete more of a solution within limited time (1 hour) than programmers using alternative control strategies.
### J.10. Movements of each Participant between Procedures

Table J.11: Movements of each Participant between Procedures

<table>
<thead>
<tr>
<th>Participant</th>
<th>Environment</th>
<th>Movement between procedures and relationship to the breadth-first global control strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Conforming</td>
</tr>
<tr>
<td>D1</td>
<td>Neutral</td>
<td>57.14%</td>
</tr>
<tr>
<td>D2</td>
<td>Supportive</td>
<td>40.00%</td>
</tr>
<tr>
<td>D3</td>
<td>Supportive</td>
<td>46.67%</td>
</tr>
<tr>
<td>D4</td>
<td>Enforcing</td>
<td>92.31%</td>
</tr>
<tr>
<td>D5</td>
<td>Neutral</td>
<td>62.50%</td>
</tr>
<tr>
<td>D6</td>
<td>Enforcing</td>
<td>100.00%</td>
</tr>
<tr>
<td>D7</td>
<td>Neutral</td>
<td>21.05%</td>
</tr>
<tr>
<td>D8</td>
<td>Enforcing</td>
<td>83.33%</td>
</tr>
<tr>
<td>D9</td>
<td>Neutral</td>
<td>41.67%</td>
</tr>
<tr>
<td>D10</td>
<td>Supportive</td>
<td>40.63%</td>
</tr>
<tr>
<td>D11</td>
<td>Supportive</td>
<td>53.85%</td>
</tr>
<tr>
<td>D12</td>
<td>Enforcing</td>
<td>100.00%</td>
</tr>
<tr>
<td>D13</td>
<td>Neutral</td>
<td>46.67%</td>
</tr>
</tbody>
</table>
References
References


References


References


References


