Supporting Design: A Computational Theory of Design and its Implementation in a Software Support Tool
by Glenn E. Holliday

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications.

APPROVED:

John W. Roach
J. W. Roach, Chair

D. Hix

D. E. McConnell

May, 1994
Blacksburg, Virginia
Supporting Design:
A Computational Theory of Design
and its Implementation in a Software Support Tool

by

Glenn E. Holliday

Committee Chair: John W. Roach
Computer Science

(Abstract)

Most work in knowledge acquisition and manipulation has focused on expert systems. Expert systems solve one kind of problem: heuristic classification. This thesis extends some advances in knowledge engineering to a broader class of problem: design.

Design is examined as a generic activity, found in many fields of professional practice. A theoretical framework is developed that supports the refinement of design from high-level concepts through implementation. This framework includes a computational model that is shown to be completely general (Turing-equivalent). Therefore, the theory and model are suitable for representing any design project. They are applied specifically to software development.

Practical support for software designers is offered in a prototype software design system. Existing work in automated knowledge acquisition is used to transfer knowledge about a design from the designer to the automated tool. Consistent support for refinement of design choices at any level of detail makes design a maintainable activity. This opens new possibilities for automated code generation, automated maintenance, and the more effective management of software at a higher-level design representation.
Acknowledgements

This work was supported in part by Navy contract N60921-89-C-A217, and in part, by Computer Sciences Corporation (CSC). A version of this paper was delivered as a contractual product to Naval Surface Warfare Center (NSWCDD), Dahlgren, Virginia. In matters of style, this thesis has benefitted from CSC’s technical reviewers. The technical content of this thesis is all my own work.

Thanks to NSWCDD Branch N86, in particular to my Assistant Contracting Office Technical Representative Robert Bartholow, and to David McConnell, who also serves on my advisory committee. I am grateful for their computing resources, and for the opportunity to complete a theoretical project that supports real design problems.

Thanks to my advisor, Dr. John Roach. He volunteered weekend hours, gave me invaluable reflections on my practice, helped me find routes around impassible obstacles, and gave much-needed reassurance that I really could get it finished. Thanks to Dr. Deborah Hix for her hours serving on my advisory committee, and for her reviews of my work.

At CSC, Robert Brown, Ken Venable, Jeff Graham, Dianna Hamlet, and Katrina Warmbrod reviewed my work and helped me balance this project with my other responsibilities.

Putting first things first, but not necessarily in that order, thanks to my partner and spouse Mariann, and to our children Jennifer, Laura, and Karen. They pushed me to accomplish the work presented here, although it really was more important to give the time to them.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>THE IMPORTANCE OF DESIGN</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Definitions</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2</td>
<td>The Place of Design in Software Development</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>PROBLEMS WITH DESIGN</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>ASSISTANCE FOR BETTER USE OF DESIGN</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Design as the Central Software Artifact</td>
<td>6</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Automated Support for Design</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>A SIMPLE EXAMPLE</td>
<td>7</td>
</tr>
<tr>
<td>1.5</td>
<td>NEW CONTRIBUTIONS OF THIS THESIS</td>
<td>12</td>
</tr>
<tr>
<td>1.6</td>
<td>PRACTICAL IMPLICATIONS</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>OTHER RELEVANT WORK</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>DESIGN THEORY</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Conceptual Maps</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Design as Constraints</td>
<td>21</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Reflection and Editing</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>PROBLEM DOMAIN MODELS</td>
<td>26</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>AUTOMATIC PROGRAMMING AND CODE GENERATION</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>KNOWLEDGE REPRESENTATION</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>KNOWLEDGE ACQUISITION</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>PROBLEM ANALYSIS</td>
<td>34</td>
</tr>
<tr>
<td>3.1</td>
<td>THESIS GOALS</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>PROBLEM DESCRIPTION</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>ISSUES AND STRATEGIES FOR SUPPORTING DESIGN</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The Design Process</td>
<td>39</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Requirements for a Design Theory</td>
<td>40</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Problem Representation</td>
<td>40</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Design Goals</td>
<td>41</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Bridging from Problem to Solution</td>
<td>42</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Communication with External Interfaces</td>
<td>44</td>
</tr>
<tr>
<td>3.3.7</td>
<td>Metaknowledge</td>
<td>45</td>
</tr>
<tr>
<td>3.3.8</td>
<td>Code Generation</td>
<td>46</td>
</tr>
<tr>
<td>3.3.9</td>
<td>Applicability to Actual Software Projects</td>
<td>47</td>
</tr>
<tr>
<td>3.3.10</td>
<td>Intended Audience</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>PROBLEM CONTEXT</td>
<td>48</td>
</tr>
</tbody>
</table>
## Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>ON SELF-REFERENTIAL PROJECTS</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>A COMPUTATIONAL THEORY OF DESIGN</td>
<td>51</td>
</tr>
<tr>
<td>4.1</td>
<td>META-THEORY: PROPERTIES OF A THEORY OF DESIGN</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>AN INFORMAL THEORY OF DESIGN</td>
<td>54</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Problem Setting and Problem Solving</td>
<td>56</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The Process of Design</td>
<td>59</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Objects of a Theory of Design</td>
<td>62</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Operations on the Objects of the Theory</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>EXAMPLES</td>
<td>77</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Example: Data Structure Model</td>
<td>77</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Example: AEGIS Command and Decision</td>
<td>80</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Example: Meta Representation</td>
<td>81</td>
</tr>
<tr>
<td>4.4</td>
<td>FORMALIZATION OF THE THEORY</td>
<td>81</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Interpretations and Analogies with Other Representations</td>
<td>83</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Structure and Components of the Model</td>
<td>84</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Values, Domains, and Types</td>
<td>87</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Annotating the Conceptual Map</td>
<td>88</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Semantics of the Conceptual Map and Goals</td>
<td>97</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.6</td>
<td>Computational Meaning of Constraints</td>
<td>98</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Choice, Loops, and And/Or Structure</td>
<td>101</td>
</tr>
<tr>
<td>4.4.8</td>
<td>Semantics of a Constraint</td>
<td>105</td>
</tr>
<tr>
<td>4.4.9</td>
<td>Semantics of a Concept</td>
<td>106</td>
</tr>
<tr>
<td>4.4.10</td>
<td>Computations on the Map</td>
<td>109</td>
</tr>
<tr>
<td>4.5</td>
<td>COMPUTATIONAL COMPLETENESS OF THE MODEL</td>
<td>114</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Formal Description of the Turing Machine</td>
<td>116</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Formal Description of the And/Or Graph</td>
<td>120</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Equivalence of the Descriptions</td>
<td>123</td>
</tr>
<tr>
<td>5</td>
<td>THE DESIGNER SOFTWARE TOOL</td>
<td>129</td>
</tr>
<tr>
<td>5.1</td>
<td>DESIGN OF DESIGNER</td>
<td>131</td>
</tr>
<tr>
<td>5.2</td>
<td>KNOWLEDGE REPRESENTATION</td>
<td>133</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Blackboard</td>
<td>134</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Prolog Factbase</td>
<td>134</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Concept</td>
<td>134</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Relationships among concepts</td>
<td>137</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Constraints</td>
<td>138</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.7</td>
<td>Domain Knowledge about Design</td>
<td>140</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Domain Knowledge about Prolog</td>
<td>142</td>
</tr>
<tr>
<td>5.2.9</td>
<td>Edit Actions</td>
<td>143</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Loop</td>
<td>143</td>
</tr>
<tr>
<td>5.3</td>
<td>OPERATIONS ON THE CONCEPTUAL MAP</td>
<td>144</td>
</tr>
<tr>
<td>5.4</td>
<td>EDITING A CONCEPTUAL MAP</td>
<td>148</td>
</tr>
<tr>
<td>5.5</td>
<td>EDITING A CONCEPT</td>
<td>149</td>
</tr>
<tr>
<td>5.6</td>
<td>ANNOTATING A CONCEPT</td>
<td>150</td>
</tr>
<tr>
<td>5.7</td>
<td>EDITING A CONSTANT VALUE</td>
<td>150</td>
</tr>
<tr>
<td>5.8</td>
<td>EDITING A SUBGOAL BLOCK</td>
<td>151</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Evaluation of Subgoals</td>
<td>152</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Use of Variables</td>
<td>154</td>
</tr>
<tr>
<td>5.9</td>
<td>CONCEPTUAL GRAPH EXAMPLES</td>
<td>155</td>
</tr>
<tr>
<td>5.9.1</td>
<td>Sequential Subgoals</td>
<td>156</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Choice</td>
<td>157</td>
</tr>
<tr>
<td>5.9.3</td>
<td>Iteration</td>
<td>158</td>
</tr>
<tr>
<td>5.10</td>
<td>REFLECTING ON THE GRAPH</td>
<td>161</td>
</tr>
<tr>
<td>5.11</td>
<td>CHANGING A GRAPH</td>
<td>165</td>
</tr>
<tr>
<td>5.12</td>
<td>GENERATING EXECUTABLE CODE</td>
<td>168</td>
</tr>
<tr>
<td></td>
<td>viii</td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.12.1</td>
<td>Generating Code for Sequential Subgoals</td>
<td>170</td>
</tr>
<tr>
<td>5.12.2</td>
<td>Generating Code for Choices</td>
<td>171</td>
</tr>
<tr>
<td>5.12.3</td>
<td>Generating Code for Iteration</td>
<td>172</td>
</tr>
<tr>
<td>5.12.4</td>
<td>Generating Code for Other Target Languages</td>
<td>174</td>
</tr>
<tr>
<td>5.13</td>
<td>INTELLIGENT FEATURES OF DESIGNER</td>
<td>175</td>
</tr>
<tr>
<td>6</td>
<td>EMPIRICAL EVALUATION OF CONCEPTUAL MAPS</td>
<td>i77</td>
</tr>
<tr>
<td>6.1</td>
<td>EXPERIMENTAL METHODOLOGY</td>
<td>177</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Experimental Protocol</td>
<td>178</td>
</tr>
<tr>
<td>6.1.2</td>
<td>User Profiles</td>
<td>179</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Data Collection</td>
<td>180</td>
</tr>
<tr>
<td>6.2</td>
<td>RESULTS</td>
<td>181</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Qualitative User Responses</td>
<td>181</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Technical Evaluation</td>
<td>182</td>
</tr>
<tr>
<td>6.3</td>
<td>CONCLUSIONS</td>
<td>184</td>
</tr>
<tr>
<td>7</td>
<td>RELEVANCE TO SOFTWARE ENGINEERING</td>
<td>189</td>
</tr>
<tr>
<td>7.1</td>
<td>REUSABILITY</td>
<td>191</td>
</tr>
<tr>
<td>7.2</td>
<td>METHODOLOGICAL USE</td>
<td>192</td>
</tr>
<tr>
<td>7.3</td>
<td>SOFTWARE MAINTENANCE AND METRICS</td>
<td>194</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8</td>
<td>FUTURE WORK</td>
<td>196</td>
</tr>
<tr>
<td>8.1</td>
<td>&quot;WHAT-IF&quot; ANALYSIS OF DESIGNS</td>
<td>196</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Evaluating Design Alternatives</td>
<td>197</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Change Detection</td>
<td>198</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Generation of Standard Attributes</td>
<td>199</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Compute Results Beyond the Map</td>
<td>201</td>
</tr>
<tr>
<td>8.2</td>
<td>EXPERIMENTS WITH DESIGNS</td>
<td>201</td>
</tr>
<tr>
<td>8.3</td>
<td>INTERFACES TO OTHER TOOLS</td>
<td>202</td>
</tr>
<tr>
<td>8.4</td>
<td>OTHER USES OF CONCEPTUAL MAPS</td>
<td>203</td>
</tr>
<tr>
<td>8.5</td>
<td>DESIGN IMPROVEMENTS</td>
<td>203</td>
</tr>
<tr>
<td>8.6</td>
<td>CODE IMPROVEMENTS</td>
<td>204</td>
</tr>
<tr>
<td>8.7</td>
<td>IMPROVED CODE GENERATION</td>
<td>206</td>
</tr>
<tr>
<td>8.8</td>
<td>SUPPORT FOR TRADITIONAL CASE PRODUCTS</td>
<td>207</td>
</tr>
<tr>
<td>8.9</td>
<td>EXTENDING DESIGN METAKNOWLEDGE</td>
<td>208</td>
</tr>
<tr>
<td>8.10</td>
<td>MACHINE LEARNING FROM DESIGN EXPERIENCE</td>
<td>209</td>
</tr>
<tr>
<td>8.11</td>
<td>GRAPH THEORETICAL ANALYSIS OF CONCEPTUAL MAPS</td>
<td>209</td>
</tr>
<tr>
<td>8.12</td>
<td>SUMMARY</td>
<td>210</td>
</tr>
<tr>
<td>9</td>
<td>REFERENCES</td>
<td>211</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDIX A</td>
<td>EXPERIMENTAL PROTOCOL</td>
<td>217</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>EXPERIMENTAL DATA</td>
<td>219</td>
</tr>
<tr>
<td>B.1</td>
<td>DATA COLLECTION FORM</td>
<td>219</td>
</tr>
<tr>
<td>B.2</td>
<td>RESULTS</td>
<td>221</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>PROBLEM REPORTS</td>
<td>222</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>DESIGNER USER’S GUIDE</td>
<td>225</td>
</tr>
<tr>
<td>Vita</td>
<td></td>
<td>237</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conceptual Map</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Representation of a Loop in a Conceptual Map</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Levels of Refinement in a Conceptual Map</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>Structural Decomposition of Conceptual Map</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>Conceptual Map of &quot;a or b implies c&quot;</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>Map of Iteration through a Set</td>
<td>72</td>
</tr>
<tr>
<td>7</td>
<td>Structure and Knowledge Associated with a Loop</td>
<td>74</td>
</tr>
<tr>
<td>8</td>
<td>ERA Diagram of Part of the AEGIS Data Model</td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>Modelling Data Structure in a Conceptual Map</td>
<td>79</td>
</tr>
<tr>
<td>10</td>
<td>Conceptual Map of Computational Theory of Design</td>
<td>82</td>
</tr>
<tr>
<td>11</td>
<td>And/Or Tree</td>
<td>101</td>
</tr>
<tr>
<td>12</td>
<td>And/Or Graph</td>
<td>102</td>
</tr>
<tr>
<td>13</td>
<td>Structure of Designer</td>
<td>132</td>
</tr>
</tbody>
</table>

List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Growth of Concatenated String of Variables</td>
<td>126</td>
</tr>
<tr>
<td>2</td>
<td>Mapping of Formal Operations to Designer Actions</td>
<td>146</td>
</tr>
<tr>
<td>3</td>
<td>Knowledge Base of Design Constraints</td>
<td>163</td>
</tr>
<tr>
<td>4</td>
<td>Designer Prototype User Qualitative Reaction Survey Results</td>
<td>221</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

The "software crisis" continues. One contributing factor is the isolation by current practice of software design to an intermediate and transient role in the software development lifecycle. This limits and restrains the usefulness of design.

This thesis attempts to improve the understanding and practice of design in software engineering. It develops a theory that describes how design is done in any problem domain. The theory is applied specifically to the domain of software engineering to develop an automated software design support tool. The theory makes design the central activity in building and maintaining software. This approach offers benefits for reusability and maintenance of software.

The theory includes a practical framework to represent the structure of a design and to describe the values computed when the design is transformed to a problem solution. A software development support tool is described that assists design development and generates source code to implement the problem solution represented by the design.

The tool developed in this thesis, Designer, is currently implemented as a prototype. It represents and stores models of design, represents properties that are interesting to users, assists users making changes in a design, and generates source code from a design. In the longer term, it is expected to represent multiple versions of proposed changes, and to infer the effects of design changes.
Experiments conducted with Designer users gave encouraging results. The sequence of design activities described by the theory was easily understood and followed by the users. The central metaphors of exploring a problem incrementally, and mapping the concepts in the problem, were easily understood and used by people with no prior experience.

The immediate practical impetus for this thesis is the software maintenance needs of the project's customer, Naval Surface Warfare Center (NSWCDD). For several years, the Software Assurance group (N86) of NSWCDD has wanted to analyze upgrades and fixes to naval software at the design level. N86 wants to infer the impact of proposed maintenance changes before they are applied to source code, compare the expected benefits of alternate design proposals, and improve the productivity of maintenance engineers.

This thesis is also connected to, and received support from, separate tasking performed by Virginia Polytechnic Institute and State University (VPI&SU). The Prolog language system and knowledge acquisition system SITE discussed in this thesis were made available to NSWCDD for this project by earlier work at VPI&SU. Some material in this thesis, per agreement with N86, has been, and will be, made available to VPI&SU in connection with those tasks and with academic work.

1.1 THE IMPORTANCE OF DESIGN

This thesis considers design in its most general sense. It argues that design should be the central activity of software engineering. The following paragraphs introduce the basic concepts used.
1.1.1 Definitions

The literature on the theory of design is surveyed in Chapter 2. The writers surveyed there agree that design occurs in any domain that creates artifacts. A design is a plan. The process of doing design includes developing an understanding of a problem, setting as a goal some artifact whose construction will allow the problem to be solved, and developing a plan for the artifact. Metaphorically, a design is a picture of some envisioned solution to a problem.

This thesis adds to these earlier developments by adopting a computational definition of design. A design, as described in this thesis, includes descriptions of computations that, when executed, create the working results of the artifact. This is most obviously true when the nature of the artifact is computational, such as with a computer program. With other types of artifacts, such as houses, the design describes computations that, themselves, describe or simulate characteristics of physical objects. This claim draws on the field of denotational semantics, but will not be fully explored in this thesis for the general case. This thesis focuses on software artifacts.

The work of design encompasses more than problem solutions. Imagining a problem solution comes only after gaining an understanding of the problem domain and its environment. The problem in its environment is known as a problem setting. The support for design given by Designer and its conceptual map begins as a description of a problem setting, then assists in constructing a blueprint for a problem solution from the problem setting.
1.1.2 The Place of Design in Software Development

This description of design fits a very broad class of problem solving. Insights for this thesis were drawn from writings as diverse as architecture and medical practice. The immediate context for this project is a computer system, and the remainder of this thesis is focused very narrowly on the field of software development.

Software development is a collection of activities and disciplines concerning the design, implementation, and maintenance of programs to operate within computer systems. In this context, the problems are those that can be solved by computer programs. The program is the artifact to be built, and the design is largely concerned with abstract concepts of algorithms, data, and control structures suitable to computer software.

This description is too narrow for most practical purposes. Every computer program is a subpart of some larger system in which it runs. A system is a collection of hardware, software, interfaces, communications channels, human users and institutions, assembled as a larger artifact to which the computer program proper makes some contribution. System development considers not only the software, but how all of a system's parts are bound together and work together to create a problem solution.

The theory developed in this thesis addresses design in a very general sense, and is applicable to systems design as well as to software design. The Designer program can represent and manipulate concepts that refer to system components outside the computer program proper. If code is generated for complex systems, these concepts can be used to represent the interfaces between the computer program

Introduction
and the other system components. It fits well into the current recognition that a
single view of a problem is not a sufficient description for a design [25].

1.2 PROBLEMS WITH DESIGN

In 1968, an international conference sponsored by NATO coined the term
"Software Crisis" [24]. That conference was a major impetus for software
development methodologies and CASE tools. A central criticism, and diagnosis, of
the conference was "Under no stretch of the imagination can one say that Computer
Science . . . is fostering software engineering."

Speakers at software engineering conferences today, 25 years after those
words, complain that little progress has been made [44]. Various structured and
object-oriented methodologies provide heuristics for software engineers. Formal
techniques applied to specifications and code have seen unqualified successes only in
small systems [35]. Design is still a vague topic. Some genuine advances have been
made, and some improvements are claimed, both in system reliability and in
productivity. The improvements have come in small increments.

This thesis claims (in section 3.2) that one root cause of the software crisis is
that too much of the work of software development is deferred to a level of
abstraction (source code) that is too low. Humans work more easily and are more
effective at higher levels of abstraction. One route to improvement is to make design,
rather than code, the central product of software engineering. This thesis does so.
1.3 ASSISTANCE FOR BETTER USE OF DESIGN

This thesis examines the design process to produce a new theoretical model of design. The theory is used to build a prototype software design tool for practical use. The tool makes possible the maintenance of a design after its initial creation, and automatically generates source code from the design. These features encourage maintenance of the design, which this thesis argues is preferable to source code debugging.

1.3.1 Design as the Central Software Artifact

This approach is presented as an alternative to established development paradigms that make program code the most important software product. The major advantage of focusing on design is to apply the gains of structured development throughout the software lifecycle. Maintenance and "what-if" experiments can be done at a more abstract level, using fewer resources than their code-level counterparts of debugging and prototyping.

Design is examined as a computational activity. A theoretical framework is developed that supports the refinement of design from high-level concepts through implementation. This framework includes a computational model that is shown to be completely general (Turing-equivalent). Therefore, the theory and model are suitable for representing any design project. This theory is called the "Computational Theory of Design."
1.3.2 Automated Support for Design

Practical support for software designers is offered in a prototype software design system, named Designer. Existing work in automated knowledge acquisition is used to transfer knowledge about a design from the designer to the automated tool. Consistent support for refinement of design choices at any level of detail makes design a maintainable activity. This opens new possibilities for automated code generation, automated maintenance, and the more effective management of software at a higher-level design representation.

This work applies insights from Artificial Intelligence (AI) to general design activities, and specifically to software engineering. Existing work in knowledge acquisition, metaknowledge exploitation, domain theory representation, and independent software agents is applied in the design tool Designer. Previous successes in the automatic design and creation of expert systems are extended to a broader class of problem: design.

1.4 A SIMPLE EXAMPLE

This section introduces the primary tool used in this thesis, the conceptual map. A simple example is used to show how to analyze a problem setting as a set of concepts.

Consider a subset of the AEGIS Command and Decision (C&D) problem. This example is chosen because it represents the customer's problem domain to which Designer will be applied, and has been released by N86 for use in software
development outside AEGIS. No sensitive naval information is contained in this example.

One subproblem solved by C&D is to maintain a knowledge base of actions available to a ship (doctrines), and match an applicable doctrine to sensor data (known as tracks) about vehicles near the ship.

A specific doctrine is added to the doctrine knowledge base by building a data structure known as a doctrine statement record from several types of supporting data. This problem involves both mechanical construction of data structures and heuristic matching against templates of applicable doctrine templates.

An analysis of the AEGIS documentation suggests that the central concept in this problem is a base data structure known as a doctrine statement, which is then mechanically transformed to a doctrine statement record. The following concepts are used to create a doctrine statement:

a. applicable doctrine statement,
b. track,
c. operator, and
d. doctrine option.

Figure 1 is a simple conceptual map. Conceptual maps are explained in Chapter 2, and used to develop a theory of design in Chapter 4. For a first overview, this diagram can be interpreted as showing the order in which lower-level concepts are combined to create the higher-level concepts in the problem.
The concept "operator" represents an important and complex entity in the real world. It would be more fully represented by a data structure with many attributes. The supporting concept "training/tactical" is intended to model a range of values for one of those attributes, used for the current subproblem. The concept "simulated/live" is a similar supporting concept for "track".

Figure 1 contains information about how the pieces of this problem relate to each other. It shows the shape of the problem, and forms a beginning structure for a design. It is a problem statement, and says nothing about how to use the concepts in the map to solve the problem of creating a doctrine statement record. To move from
problem statement to problem solution, the conceptual map is annotated with additional information.

Each concept in the map is said to have a value. For example, an applicable doctrine statement is a set of data structures. A value is a semantic quality: the value of the concept "applicable doctrine statement" is a representation of the actual data structure that comprises an applicable doctrine statement in the AEGIS system.

The value of a high-level, abstract concept such as "applicable doctrine statement" is likely to be a complex data structure. The value of a lower-level concept, such as "training/tactical", might be an integer. The conceptual map represents a generic schema, or placeholder, for the value of each concept. While some concepts may represent specific entities with single values, many comments represent a class of possible values. When an implementation of the design is executed, it will construct concrete instances for each concept.

The conceptual map says that the value of each concept depends on the values supplied by its supporting concepts. It remains to describe how a concept combines its supporters' values to create its own value. Each concept is thought of as performing a computation, which is why the theory developed in this thesis is called a Computational Theory of Design. The design contains an algorithm for how each concept does its computation.

Because conceptual maps are a declarative representation of problem structure, it is preferable to view each computation in the design as a declarative constraint on the concept and its supporters. For the concept "applicable-doctrine-statement", the structural information in the map represents a high-level constraint that relates
"applicable-doctrine-statement" to its supporting concepts "operator" and "doctrine-option". The constraint could be stated as follows:

\[
\text{compute-from(applicable-doctrine-statement, operator, doctrine-option)}.
\]

The map is annotated with a more detailed definition of this constraint. In the following example, the constraint "template" is assumed for simplicity to generate a new data item, identified by the variable "doctrine-template":

\[
\text{compute-from(applicable-doctrine-statement, operator, doctrine-option) if}
\]
\[
\text{template(doctrine-option, doctrine-template) and}
\]
\[
\text{value-of(operator) and}
\]
\[
\text{insert-attribute(doctrine-template, operator, applicable-doctrine-statement)}.
\]

This example definition could be informally interpreted to mean that an "applicable-doctrine-statement" can be computed from an "operator" and a "doctrine-option" if a "doctrine-template" can be generated for the "doctrine-option", and the "operator" attribute can be inserted into this template.

In practice, the Designer tool supplies protocols to access the values of supporting concepts, and primitive data operations drawn from the programming language Prolog. Concepts typically are refined to lower levels of detail before their constraints can be expressed in terms of primitive concepts.

The conceptual map thus contains structural information about the problem, and has been constrained so that each concept can only generate an appropriate value. At this stage, Designer generates Prolog source code for a system to evaluate the
constraints on the map. Running the generated computer program is equivalent to executing the map to compute some final value as a solution to the original problem.

1.5 NEW CONTRIBUTIONS OF THIS THESIS

The Computational Theory of Design and the Designer tool build on the work of earlier researchers. The original contributions made by this thesis to the field include the following:

a. Conceptual maps are extended to describe not only the high-level structure of a problem setting, but also the associated problem solution. Eden's original work [7] uses conceptual maps as a tool to help a human analyst explore the relationships among concepts in a problem setting. This thesis expands the conceptual map into an annotated conceptual graph, which then provides a general theoretic framework. This is a more general and complete support for design than that discussed by Eden.

b. A theory of design is developed that supplies features not considered by previous researchers. The Computational Theory of Design extends the suggestions of Dasgupta [5] about the structure of designs in his Theory of Plausible Design. Dasgupta is most interested in maintaining the plausibility of each constraint in a design. This thesis focuses instead on describing these constraints in a way that permits the computation of values for each subgoal. This theory considers the product of design as a computation, which can be
executed to produce a value identified with a problem solution. This approach is not considered in the other work surveyed.

c. The theory also includes a formal description of the process of design. While the literature includes other formalizations of parts of the software development process [27], the formalization of this thesis is used directly as a basis for an automated program to perform some of the activities humans do during design.

d. The conceptual graph described in this work supports refinements of a design at different levels of abstraction, in a common theoretical framework and knowledge representation. Older models refine a design hierarchically from high levels of abstraction to low levels of detail. The Computational Theory of Design provides a uniform treatment and a smooth transition from the highest level abstractions all the way to the implementation of source code.

e. This thesis introduces a simple, intuitive metaphor of editing to explain the theoretical description of the process of creating a design. This is an extension of Schön's [34] description (in contexts more general than design) of a cycle of act - reflect - extend the theory of a problem domain. The program Designer also uses the metaphor to implement most of the interaction between designers and their designs as a set of editors.

f. This work includes a formal definition of the boundary between problem setting and problem solving. The fuzziness of this boundary is a source of ambiguity in some software engineering methodologies.
The Computational Theory of Design identifies it with the difference between identifying general constraint rules and instantiating specific instances of those constraints.

g. One subproblem addressed in this work is an appropriate language for describing the constraints that hold on a set of design concepts. This language is made more declarative by using the structural relationships of the conceptual graph to express choice and iteration. This feature is inspired by theoretical treatment of and/or trees, but is an original contribution to actual support for practical design.

h. The Designer tool uses AI techniques to supplement human strengths, using cooperative design agents to help connect products of top-down and bottom-up design actions.

1.6 PRACTICAL IMPLICATIONS

Better support for the activity of creating and considering software design can improve many software engineering activities, including the following:

a. The measurement and comparison of alternate design ideas and strategies,
b. Documentation and traceability of design decisions,
c. Automatic generation of code,
d. Provability of some classes of correctness and consistency claims, and
e. Maintenance of installed systems.
Problem-solving in any domain proceeds more effectively when it can be done at a more abstract level. Abstracting a problem creates a smaller, simpler problem.

This thesis emphasizes the higher levels of design because of the potential gain in productivity and reliability if designers could work at that level for more of the software development lifecycle. The longer software engineers can avoid writing source code, the easier it will be for them to work with the system itself, rather than with the syntax of the implementation language. Requirements and design became the focus of software engineering in the 1970s for the same reason: they allow planning and reasoning about a software system at a higher level of abstraction than do programming languages.

The tool described in this report attempts to replace implementation syntax with design semantics. Although every representation has a syntax, the Designer tool uses one that is intended to be simple, natural, and declarative. It encourages designers to continue describing the relationships and constraints in the problem space, rather than concentrating on procedural, code-like descriptions. It supports every development activity, from concept development through post-delivery maintenance. Designer generates source code from a design automatically. The software design can thus replace source code as the most important interface to the human developer. Among other potential benefits, this thesis attempts to turn code maintenance into design maintenance.
CHAPTER 2
OTHER RELEVANT WORK

This thesis builds on, and draws insights from, earlier work in design theory, automatic programming, and several fields of AI. The most relevant work in the literature is reviewed in this chapter.

2.1 DESIGN THEORY

The earliest design theory was found in the literature of architecture [17]. These sources interpret design as the creation of structure, to be used later to realize a constructed object (an artifact). This is a fundamental definition that is echoed throughout computer science literature.

Various approaches to design theory were found in the literature of hardware design, fault diagnosis, and industrial product design. A significant and early work in the computer science literature is Simon’s Sciences of the Artificial [39]. Simon’s central insights are that design has structure, and that problem statements are transformed into problem solutions.

Simon argues that problem solutions often emerge from a well-structured, formal problem statement. Designers gradually refine a problem description until it becomes a description that the designer recognizes as resembling a desired solution. Although he ignores the distinction between problem setting and problem solution, this iterative process is key to a theory of design.
In (Simon's) other words, design is representing a problem so that the solution becomes obvious. This could be interpreted as a claim that a solution is equivalent to a sufficiently detailed problem setting.

He also describes design as search through a large problem state space. The states in the problem space are partial descriptions of alternate designs. The "leaf nodes" in the problem space seem to be synonymous with alternate problem solutions.

Simon does not develop a specific theory, but offers guidelines for the development of a theory of design. This work is frequently cited in the theoretical branches of computer science. Simon later updated some of his views in [38].

Most current theories about design appear in formalizations of the software engineering process or lifecycle [27]. These formalizations assume a general pattern in which designers recognize a problem, choose goals and constraints to be met by a solution, and refine those constraints into a detailed plan for implementing a solution. In software systems, implementing a solution involves building a software artifact that computes the desired solution.

Moving from problem to solution involves creating a plan (the design) for an artifact that can transform the environment to a defined solution state. The plan describes an imagined future state. The plan typically includes formal requirements and constraints that the solution artifact must meet. The plan is then followed to create the desired artifact, and the artifact is employed to solve the problem.

All these traditional theories assume that software development moves through a sequence of phases. Requirements development is assumed to be a different type of
activity from design, and design from writing code. This thesis departs from that assumption, and adopts alternate insights from other writers surveyed in this chapter.

2.1.1 Conceptual Maps

The major work used by this thesis is Eden et. al.'s conceptual map, described in Messing About in Problems [7]. Its central idea is that designers do not create a monolithic description of a single, large problem. Problem analysis is difficult because it typically deals with subjective problem messes (Eden's term). A designer needs to explore and "mess about" with the pieces of the problem in order to conceptualize gradually its larger structure. Eden also claims that there is no such thing as an objective problem. Designers therefore need a tool to reorganize a problem and show it from multiple points of view.

Eden introduces the conceptual map to incrementally organize and work with a partial problem description. The map is a picture of the concepts that the designer discovers are relevant to the problem. A concept for Eden is similar to an object or entity in traditional systems analysis. It represents the ideas - literally, the mental concepts - with which a designer describes a problem. Concepts are typically abstract, high-level notions in the early stage of a problem analysis.

A set of design concepts is represented as a graph of nodes (representing concepts). Eden associates each concept with a pair of values, representing opposing extremes. For example, a concept representing "track speed" may be associated with the value pair "low - high".
A conceptual map uses arcs on the graph to represent support relationships between the concept nodes. Concepts belong together in a problem description when they have some relationship to each other. A concept is said to be supported by another concept when that other concept in some way influences its value. For example in the conceptual map of Figure 1, the concept "doctrine statement" is supported by the concept "track". This is interpreted to mean that a "track" is required, or used in some way, for a "doctrine statement" to exist.

Intuitively, it is not surprising that conceptual maps usually include loops. Complex systems often include feedback mechanisms. For example, feedback loops form the central dynamic paradigm of dynamic system modelling. For Eden, they determine whether the problem setting is likely to be stable. In an example from Eden, many concepts in a publishing business are influenced by readership. Yet those concepts are also likely to influence readership. This example is illustrated in Figure 2.

![Figure 2: Representation of a Loop in a Conceptual Map](image)

Eden develops an informal theory about how people create and organize concepts. Eden does not formalize these ideas, nor does he give a rich enough set of
operations on a conceptual map to work with general computational designs. Eden concentrates on understanding the problem. He gives little consideration to problem solutions. Eden's context is the work of human consultants. The human part of the task, for him, is seeing a problem solution emerge out of a well-done problem setting.

_Messing About in Problems_ describes a computer program to maintain and report on a conceptual map; however, it is only a sketchpad for a human user's analysis of the problem. This work was used as a starting point for a theory of design. The Computational Theory of Design described in this thesis makes many extensions to Eden's central paradigm.

At least one software development tool, KAT [30], implements concepts similar in appearance to Eden. That tool uses dynamic system modelling as its theoretical foundation, but drops its quantitative feedback to produce an influence graph very much like a conceptual map. The author makes no references to Eden.

SITE [28] is an earlier project at VPI&SU that extended conceptual maps by annotating them to include concrete representations of data in the problem domain. Where SITE generated expert systems that solved heuristic classification problems, Designer represents more general problems. It generates code for systems that construct original solutions. Designer also differs from SITE in its support of incremental refinement and maintenance, by building multiple layers of detail in a design.
2.1.2 Design as Constraints

Dasgupta, in Design Theory and Computer Science [5], makes a major recent contribution to design theory. He identifies a design with a network of constrained nodes, each representing some subpart of the design. Dasgupta agrees with Eden [7] and with Schön ([34], discussed in section 2.1.3) that a designer does not create a single design description by translating a monolithic, perfectly known requirements description. Rather, design proceeds incrementally. At each moment in time, a design should be in a plausible state. "Plausible", for Dasgupta, means that each node in the design network has appropriate and believable support.

Dasgupta’s "Theory of Plausible Design" provides a framework for representing and incrementally modifying a design. He develops a calculus to refine design constraints. The calculus computes the plausibility of each design node, leading to a truth maintenance system. The network of constraints also maintains a history of decisions made as the design changes over time.

Dasgupta observes that each design decision has a goal, representing some subpart or milestone toward the overall solution represented by the design. It is the history of decisions that provide plausible support for each design goal. The nodes in Dasgupta’s networks represent design goals; the arcs between them represent constraints of one goal on another.

Dasgupta identifies several purposes for design; the central purpose for this project is to be a blueprint. Some actual artifact (software system) will be implemented by making its structure conform to the design blueprint. The essential character that distinguishes a design from other types of problem descriptions is the structure it imposes on the problem and solution spaces.
The significance of structure, for Dasgupta, is that it expresses constraints among the concepts of the design. This leads Dasgupta to a richer data structure to support his design structure. In Eden, the conceptual map is flat. Dasgupta also uses a directed graph to represent a design. His design network is layered, and lower-level nodes support higher-level nodes.

Dasgupta claims that there is no qualitative difference among requirements, design, and implementation. This thesis adopts that insight as central. These three traditional phases of software development are convenient labels for levels of abstraction on the path from highest-level problem statement to implemented artifact. In reality, there may be arbitrarily many levels of increasing detail on that path. Some specific concept may not belong to any of the three phases of development. Knuth's work in literate programming [18] suggests that this principle is also true of source code implemented from more traditional designs.

This thesis builds on several elements of Dasgupta's work. It uses his application of constrained networks to formalize the conceptual map as a graph data structure. Dasgupta's network of constraints is modified to create computational annotations to the concept nodes of a conceptual map. Where Dasgupta introduces problem solving at the very beginning, Designer keeps the focus on incrementally updating the problem setting. This thesis includes (in section 4.2.1) a clear definition of the difference between problem setting and problem solving.

The idea that design is a smooth transition across many levels of detail from high-level abstractions to implementation suggests that code generation should follow naturally from design. Dasgupta is more interested in the truth maintenance of a design than in generating additional products from it. Designer (like SITE before it)
automatically generates Prolog code to implement the software artifact described by a design.

Stefik's work is also relevant here [42]. He works directly toward the goal of a problem solution. For Stefik, a solution is defined by the constraints on it. He believes that decisions to instantiate specific constraints should be deferred as long as possible. This helped clarify the definition given in section 4.2.1 that defining specific instances of constraints mark the boundary between problem setting and problem solving.

Stefik also separates a problem into levels of detail. This idea was expanded later by Simon [38], and is also used by Designer.

2.1.3 Reflection and Editing

Schön's The Reflective Practitioner [34] examines how practitioners of various professions use formal knowledge in their problem domains, and extend it by informal actions. His interest is broader than design, but his insight is an important bridge between anecdotal and formal approaches to design.

Schön presents experimental evidence suggesting that human designers do not plan the details of a solution, but recognize deficiencies in a problem state or partial design, and make refinements to remove the deficiencies. The central human cognitive activity may be not logical planning, nor reasoning at all, but non-reasoning pattern recognition. Schön argues that problem solvers do not rationally know just what the problem is.
He describes how professionals make small, incremental changes to a situation, without knowing completely how those changes will affect the total environment. This is not a formalization, but it is a useful model for human problem solving. In Schön's model, humans move a problem toward a solution by nudging it, a little at a time. The model is not explicit in Schön, but its fundamental operations seem to be as follows:

a. Select a subpart of the problem description.
b. Change some limited part of the problem description.
c. Determine the impact of the change on the surrounding problem description.
d. Reflect on the results of the action.

This is a good description of editing. Rephrasing Schön, editing can be taken as a fundamental human activity. This lends additional support to the notion that a design tool should be, in essence, an editor on a problem space. Schön, in common with many writers, does not define clearly the crossover from problem setting to problem solution. Both activities use the same types of concepts, and therefore need the support of the same type of conceptual editor.

Schön's fourth step, reflection, is a meta-step. Reflection is outside the operations on a specific design. The purpose of reflection is to create new knowledge about the task in process. This knowledge presumably makes some new connection or recognition about the task. The designer thus uses reflection to extend the theory about the problem domain. This improves the ability of the designer to create future designs.
Schön describes a general pattern of human problem solving. He describes professionals alternating between "reflection-in-action" (his term implies using compiled knowledge without thinking about it) and reflection on action, after the action is complete. People applying any skill to a problem tend to act on the problem iteratively, to reflect on the results of their action, and to use their understanding of the results to include some aspects of the problem within the well-understood practice of the skill in use. Chandrasekaran [3] also analyzes observations of designers in these terms.

Most of a domain is not initially described by theory. Both theory and practice in that domain must understand and account for this situation. However, it is still desirable to move new pieces of the domain into the formal description of the theory. It can then be managed by repeatable procedures, which are less costly than ad hoc attacks on novel situations that are not well understood.

Schön describes this extension of theory as the attempt to move from craft to science. Scientific knowledge is a body of consistent facts about the usual situations encountered. He describes the well-known, repeatedly applied theory as an island of high ground, surrounded by a swamp of special cases and novel circumstances. Practitioners must rely on craft, which often means non-rational problem solving, to handle problems outside the theory. Because of the complexity of most domains, practitioners encounter the special cases more often than they apply the central domain theory. Every professional domain therefore seeks to extend its own theory, moving more of its experiences into the type of knowledge that practitioners can handle most easily.
2.2 PROBLEM DOMAIN MODELS

Several researchers have built programs that help a human designer build a program by working at the design level. A series of projects at Carnegie-Mellon University, and others inspired by those projects, depend on building a model of a specific problem domain. Examples of this approach include SALT [21], MOLE [9] and ASK [12]. These programs use their domain knowledge to suggest design choices to the user. The product built by these programs is used to design other end products.

For example, SALT was used to build an expert system that advises engineers who design elevators. SALT acquires from its user a knowledge base about the parts and plans used in elevators. The program that SALT generates searches this knowledge base to find the correct structure for an elevator. The problem is solved by classifying its subparts, and matching them against libraries of solution structures. SALT, and the other programs in this group, solves a heuristic classification problem in a specific problem domain.

SALT and its descendants share a problem solving paradigm known as propose-and-revise. Its basic strategy is to start from a standard partial solution, and supporting its refinement and editing. This is very similar to the standard practice in engineering. An engineer selects a standard method from a handbook appropriate to the problem domain, then tailors it to fit the specific problem.

Designer shares this strategy, but implements it differently. SALT is a planner tool, and uses its knowledge of a finished design structure as a goal. MOLE works forward from a partial design by using its domain knowledge to find opportunities to propose new additions. Designer does not know what concepts
should be in a finished design, and does not plan for the finished design. It evaluates a partial design against a knowledge base of general design constraints, rather than using knowledge about a specific target application domain. Rather than building a domain model for use by other experts, Designer supports the direct development of systems that may be intended for any target domain.

SALT, like Designer, constructed problem solutions. This distinguished it from other classificatory systems. SITE [28] also constructs expert systems to solve heuristic classification problems. Designer generates systems, like these two do, but not expert systems. Designer supports design and generates code for systems that perform more general computations.

Design is used in these systems as planning knowledge, to control the acquisition or retrieval of knowledge. For example, SALT [21] uses knowledge about how to design a class of artifacts in order to plan what specific knowledge it must acquire to refine that knowledge to a specific design.

Molgen [42] is an example of a planning system that produces a synthesized product, similar in some ways to the product of a design. Molgen is a very specific knowledge base for a specific problem domain. It is interested more in multiple control levels for constraint propagation than in the general problem of design.

Molgen’s multiple control levels are related to Simon’s model of multiple problem spaces [38]. This conceptual metaphor is also seen in Dasgupta’s incremental refinements [5], and in Designer’s refinement of multiple levels of detail in a conceptual map. These are all ways to support top-down design, and iterative refinement of abstract models to more detailed ones. See Figure 3 for an illustration of problem spaces implemented as levels of a conceptual map.
Shafer in [36] addresses some meta-theoretical problems relevant to this work. He observes that it is difficult to develop formal models that represent adequate quantities of reality at useful levels of detail. He calls this phenomenon "model aliasing." It suggests that formal descriptions of rich, complex human activities are always in danger of inadequately describing their targets.

Shafer's solution to this problem is model coherence: the property in a model of including redundant information in the areas of interest. This is accomplished by choosing primitive modelling elements that are not too primitive, but can contain rich and complex information about a specialized part of reality. The structure and
operations of the model can then be kept simpler than the elements they order and manipulate. The Computational Theory of Design applies this principle to choose its fundamental model elements, as discussed in section 4.2.4.

Chandrasekaran [3] also writes about meta-issues for modelling. Much of his work concerns finding the right level of abstraction for problem representation. His ideas were a generic influence, though this thesis uses none of his concrete contributions.

2.3 AUTOMATIC PROGRAMMING AND CODE GENERATION

Early design-related work is found in efforts toward automatic programming. The Programmer's Assistant project [33], done at MIT, is especially relevant. Barstow [11] summarizes work in automatic programming.

The mainstream of this tradition is oriented to domain knowledge in specific language environments. Some projects have developed theories of what humans do when they code. Most of them emphasize a robust knowledge base for transforming procedural steps into the assumptions and structures of a target language. Automated programming usually relies on a large amount of knowledge about two different domains: the target programming language domain, and the problem domain for which a system is a solution.

The Computational Theory of Design is a more general theory of what humans actually do when they design. The program developed from the theory does not do automatic design so much as assisted design. Its most important knowledge domain is not the details of the target programming language, but the general theory of the

Other Relevant Work
process of design itself. This knowledge is not tied to specific design methodologies. It is a theory of concepts and how to connect them. Rather than knowing a repertoire of strategies for translating a loop into syntactic structures, the theory understands how to recognize a loop.

Designer generates code; therefore, it must have a domain theory for the target language. Because the theory of design developed here is a computational model, Designer uses a target language (Prolog) whose computational model is very similar to that developed in the Computational Theory of Design. This minimizes the amount of language domain knowledge needed. Designer therefore owes more to standard compiler theory than to classical automatic programming.

A deeper understanding of why automatic programming seems so like and also so unlike this project originated from the work of Chandrasekaran [3]. His model of generic tasks suggests that the theory of automatic programming could be lifted to a more generic level. Automatic programming is an example of a generic task. The theory of design pursued here treats design as a generic task. The two tasks share many goals and superficial structure. However, the two tasks are accomplished by very different methods.

2.4 KNOWLEDGE REPRESENTATION

Much work in knowledge engineering seeks generic knowledge representation for useful domains, in support of large projects such as natural language understanding and integrated, independent agents. Systems that emphasize knowledge representation typically develop a theory of knowledge manipulation, and a repertoire of operations to transform knowledge structures.

Other Relevant Work
Because this tradition is so important in AI research, the literature describes a large variety of knowledge representation architectures. These were surveyed for representations applicable to design. Although the conceptual map is used as the basic knowledge representation, additional expressive power was needed to handle several types of knowledge not included in Eden's original work.

Very few references in the literature consider design as a type of knowledge manipulation. Most of the ones that do so are concerned with the design of industrial products [39], or equate design with building procedural plans for problem solving [41]. Several knowledge representation techniques originally used in other problem domains were applied to this thesis.

The knowledge structures used in this work were partly inspired by semantic networks [45]. The annotations that this theory adds to conceptual maps are influenced by denotational semantics [43], by attributed grammars in programming language theory [1], and by various frameworks for general reasoning [38] which represent general knowledge in semantic networks.

The graph notation for conceptual maps is syntactically like that of semantic networks [45]. The semantic interpretations of the two kinds of graphs are also similar. In each, nodes represent things abstracted from a problem domain, and arcs represent relationships between the nodes. The fundamental difference is that a semantic network is typically given a much more specific, formal, and detailed semantics.

Conceptual maps and semantic networks serve different purposes. For Eden, a conceptual map helps a human analyst recognize patterns and relationships in the problem structure. Arcs are given a general and informal meaning, which is
sufficient to support a human navigating the problem setting. Concept nodes may represent specific physical entities, but more commonly represent high-level, abstract concepts.

Semantic maps typically support automated reasoning on the things represented in the network. Both nodes and arcs in a semantic network are assigned specific and detailed meanings. A type or class hierarchy is often supported, and it is common for nodes in a semantic network to be associated with a large quantity of knowledge about specific objects or entities. Although the nodes in a semantic network can represent abstract concepts, it is common for a semantic network to contain a larger amount of information about concrete, real-world entities than does a conceptual map.

Clearly these differences are in the amount of information contained in the two types of graphs, not in their structure. Although Eden does not do so, it would be fair to class conceptual maps as a more abstract variety of semantic network. The extensions added to conceptual maps in this thesis make them look more like semantic networks than does Eden's original description.

Conceptual maps are also closely related to dynamic system modelling [10], which is itself related to semantic networks, but do not support quantitative, continuous simulation. The basic formalism of the conceptual map denotes the ideas devised by designers, and the structure of a problem. The values associated with concepts can be used to compute relative values of supported concepts, but this is not directly supported by Eden's use of the map. The conceptual map is not a finished model to support dynamic simulation, but a way to support change to the model itself. The concepts found in a problem are linked and unlinked into various patterns and structures.
Other work in formal knowledge models, such as Inductive Logic Programming, is likely to be relevant, but was not surveyed until after the work described in this thesis.

2.5 KNOWLEDGE ACQUISITION

The heuristic-classification expert systems, particularly ASK and other work by Gruber [12], share an emphasis on knowledge acquisition. The older tool SITE, which falls squarely inside the knowledge acquisition tradition, was used as a starting point for the design tool Designer. Designer uses knowledge acquisition techniques, but is not a knowledge acquisition system. Knowledge acquisition provided important insights and suggestions concerning identifying the tasks and operations of design. Knowledge acquisition is a way for either a human or an automated program to record domain knowledge that will be used in a completed system. Design theory, and the Designer program, are more interested in managing and manipulating that knowledge after it is acquired.
CHAPTER 3
PROBLEM ANALYSIS

This chapter discusses how that desire for better design support was refined to more specific project goals. It identifies the major issues, assumptions, and problems that were discovered during this project.

3.1 THESIS GOALS

This thesis began as an effort to maintain a design through the life of a software product. If the hard work of maintaining a design could be automated, it might become feasible for the design to continue, and be used, through the lifecycle of a system. This would solve many of the software engineering problems discussed in section 1.2.

The most general goal of this thesis is to promote design to the central product of software development. This immediately suggested the subgoal of finding effective ways to automatically maintain designs.

The problem was originally posed as a metaknowledge question: what knowledge does a design need about itself to maintain itself? This led to an investigation of design theory. The original goal refinement of self-maintaining design was quickly replaced by the following set of subgoals:

a. understand how humans do design,
b. develop automated support for human design activities, and
c. develop automated versions of some of these activities.
These three subgoals quickly implied two others:

d. develop a theory of design, and

e. develop a knowledge representation to support design activities.

3.2 PROBLEM DESCRIPTION

Currently, source code is treated as the most "real" product of software development. A common proverb among software engineers is "the true design is in the code." Only the code is consistently maintained as needs change. This is because of the expense and difficulty of maintaining additional design documents in parallel with code. Maintaining each product is expensive. Keeping the two synchronized is an additional layer of complexity that adds cost to maintenance. These factors have made design documents a luxury for most software projects -- especially after the initial delivery.

When the design is "lost" in this way, the advantages of working at a higher level of abstraction than the code are also lost. Schön observes [34], and Simon discusses the experimental evidence supporting the claim [39], that humans work better at more abstract levels. These writers support a fundamental assumption that it is desirable to organize any human activity so that as much of the work as possible is done with more abstract views of the problem domain.

From this assumption follows the goal of making design a "more real" product of software engineering than its current status. If design becomes the central product, then source code becomes something of an appendage to a design.
Source code demands significant attention because it is not simply a translation from a design. Coding uses unique skills and a large knowledge base about specific computer languages. The relative status of design and code can only be achieved when it becomes less costly to work at the design level than at the code level. Effective automatic code generation would make this so. This is the meaning of the colloquial goal of CASE tools to "make coding obsolete." If design is where the work of software development is actually done, current skills and knowledge about both development and maintenance could become design knowledge. The advantages of working at the more abstract design level would apply throughout all phases of the software development lifecycle.

Automated CASE tools address this problem, but have made only incremental progress toward solving it. Most CASE tools help users create a design document, but give little or no support for maintaining that design over time. The three parallel descriptions of requirements, design, and code are maintained separately, often with the aid of separate CASE tools, and therefore at great cost.

Transforming higher-level notions of system requirements and design into specific implementations is difficult. It is commonly accepted that the greatest number of errors and the most serious errors are introduced into systems at these higher levels, but discovered at the lower levels. In software, design remains predominantly a manual art. Automated support ("CASE tools") generally handles only bookkeeping tasks. Users can create descriptions of systems in several different ways (commonly in graphical forms), and maintain these descriptions in CASE databases. Automated tools routinely generate source code data declarations from designs of data structures. Procedural algorithms and other aspects of designs have proven more difficult to generate automatically. Designers commonly use ad-hoc combinations of elements from several methodologies.
There are advantages to ad-hoc methods. Software is like other complex systems, in that the system cannot be clearly seen if all its details are presented in one model. For example, in architecture, structural plans are separated from mechanical drawings and material specifications. In software, the structure of the system is a type of information very different from the contents of each element in that structure. Each of the design methodologies addresses only a subset of the many views [25] of a software system.

Among the possible causes for the difficulties of software engineering are the following:

a. difficulty conceiving a correct requirement,
b. the complexity of separating and connecting the multiple views that are constructed at different levels of abstraction during different phases of software development, or
c. the complexity of transforming high-level views to low-level code.

This work addresses all three issues. It offers a common representation for views at arbitrary levels of abstraction. It partially automates the work of transforming high-level concepts to low-level code. The automated design assistant described here captures and models the structure of concepts and design. It cannot judge the correctness of the semantic contents of design concepts. It does, however, give a uniform structure for a large set of design concepts. It associates design descriptions with design computations. It supplies not only automated bookkeeping for the incremental construction of a design, but automated review, evaluation, and extension of the design. Finally, it generates source code to implement the computations of the design.
Despite its early promise, the benefits of CASE technology have not improved productivity to the level desired [8]. Most current tools assist the manual details of low-level work. Much more might be gained by using the more fundamental tool of abstraction. The literature has moved to advocacy of higher-level systems analysis ("enterprise modelling"). The CASE tools that address this level treat enterprise knowledge as another view, often described using a different syntax than data definitions.

This thesis assists the way developers think about their design. A design is approached, first, as a collection of concepts that humans use to describe a problem domain. All work connected with a problem description is kept at the highest, most abstract level possible for as long as possible.

3.3 ISSUES AND STRATEGIES FOR SUPPORTING DESIGN

The theoretical writers discussed in Chapter 2 provide insights about specific concepts in this problem. None cover all of the design needs discussed in Chapter 1. In organizing the existing material to find support for each of the project goals identified in section 3.1, a number of major needs and themes were identified. They are discussed in the following sections.
3.3.1 The Design Process

A goal of this thesis was to create a computer program that supports design. The project uses an AI paradigm: emulate a human activity in an automated program. It adopts several assumptions associated with this paradigm:

a. An automated program can assist or replace a human in some activity by imitating some or all of the steps that humans perform in the activity.

b. To emulate a human activity, a program must have some understanding of the activity. This requires a well-developed understanding of how humans do the activity.

c. A sufficient understanding and description of the subject domain is expressed as a theory (in this case, a theory of design). An informal theory can assist human understanding, but a formalization of the theory assists its validation.

d. A formal theory can be implemented in a formal computational tool (a computer program) to perform the actions described by the theory.

To this end, observations of human design activities were gathered from the literature discussed in Chapter 2. A general theory of design was developed, which is documented in Chapter 4. The tool Designer described in Chapter 5 is (largely) a restatement of this theory as a Prolog program. The activities that are best understood (most fully described) by the theory, such as recording knowledge in the conceptual map and searching for inconsistencies, are most completely automated in
3.3.2 Requirements for a Design Theory

Recognizing that a theory about design is needed leads to questions about what kind of theory. The central knowledge representation, the conceptual map, is intended to be an analog model of the concepts it represents. A model-theoretic approach was chosen as a good match to this feature. The intention to translate the theory directly to a concrete form as a Prolog program implies a level of formality that supports a rich description, but relies on an intuitive semantics. These issues are discussed further in section 4.1.

3.3.3 Problem Representation

A generic representation for problem settings is needed. This work assumes that a single representation can be found that is suitable for any design problem in any arbitrary problem domain. That assumption suggests that a suitable representation will be simple and general, both in its syntax and semantics.

Although there are well-developed methodologies for representing components of designs (e.g., Entity-Relationship-Attribute and Object-Oriented models), no such model was found for the process of design. It was assumed that a tool that could provide the support described in section 3.1 would need to infer the effects of design actions. That implies that the tool needs to understand the design process. This
understanding of how design is done was missing from available models and modelling tools.

The conceptual map was chosen as a good fit to these needs. It was developed by Eden to support his own informal theory about how humans work with problems. Both the map and Eden’s theoretical beginnings were used as the basis for later enhancements and extensions.

It was necessary to assume that the very generic "supports" relationship expressed in the graph arcs is sufficient to express and work efficiently with both processes and data. This assumption is partly justified by previous work reported by Eden [7] and Reddy [28], but needs to be measured against later empirical experience.

3.3.4 Design Goals

According to Eden [7] and Schön [34], human designers do not start out with a clear, rational understanding of a complete problem or a proposed solution. However, as they evolve a better problem understanding, their design goals also become clearer.

The goals of a design are not represented explicitly in this work. Design goals are implicit in the higher-level concepts of a conceptual maps (specifically, concepts that support no others are presumed to represent high-level goals). This is similar to the implicit encoding of goals in most procedural programming languages. This approach followed from the assumption that the crux of the design problem is to represent and manipulate a changing problem setting.
The theory developed in Chapter 4 describes how to automatically infer a unique design goal from a conceptual map. The Designer tool can create a default conceptual map that includes a unique goal concept named "top-level-concept". It became clear in the late stages of this work that the user’s goals should have been made explicit. A goal declaration would assist the user’s view of the problem setting, and be more consistent with the declarative nature of the conceptual maps.

An explicit design goal is naturally supported by a conceptual map. Intuitively, some high-level concept can always be constructed that represents the overall goal for a software development project. Section 4.4.5 develops a formal support for this intuition. A conceptual map can naturally represent a top-level goal as a distinguished concept node. This distinguished node is analogous to the start state of a finite state automaton.

3.3.5 Bridging from Problem to Solution

This work assumes that it is best to keep design work at a high level of abstraction for as long as possible. This "highest level" is identified with work on the problem setting. Despite the desire to do without distinctions among analysis, design, and code, it became clear that the designer eventually moves away from a pure problem setting to create a problem solution.

As the problem setting was identified with Eden’s conceptual map, concerned mainly with the structure of a set of concepts, so the problem solution was identified with Dasgupta’s constraint network. The desire for a simple, consistent problem representation led to the recognition that a constraint network could be easily
represented as a conceptual map, augmented with constraints on each concept and its supporters.

A method was needed to transform one domain to the other. The similarity of the original conceptual map and the annotated conceptual graph suggested that it should be possible to automate the transformation.

Design constraints are represented explicitly as annotations attached to each concept in the map. Designer includes an editor to help users express these constraints. Each constraint can be interpreted as defining a value for the concept to which it is attached. Although Designer does not understand the semantics within the constraints, it does give automatic assistance for syntactic evaluation and extensions to the transformations from problem structure to design constraint.

The form of the annotations was an important question. The presumption was that declarative is better. The entire computational model of both theory and tool was shaped by that of Prolog. That decision was, in turn, based on the assumptions that Prolog was appropriate both as a target language for code generation and as an implementation language for the formalizations of the theory. Another presumed advantage of declarative representation is that it directs the user's attention to a logical description of what the constraint is. This is assumed to be preferable to procedural ways of considering a constraint.

The features desired in a constraint annotation were also influenced by SITE. That tool attached action frames to a conceptual map, which supported a limited number of procedures and operators. This thesis assumed from the beginning that the ability to describe completely general computations was important. That assumption requires that the constraints be expressive enough to express general computations -
i.e., be equivalent to a Turing machine. This requirement for computational
generality is the reason that section 4.5 presents a proof that it was achieved.

All these factors led to the choice of a general logical predicate, based on
Prolog's Horn clause semantics, as a constraint representation. However, a
conflicting need was felt to provide end users with a familiar syntax. Designer
therefore allows users to express themselves in procedural assignment statements, then
translates these to an internal logical clause form. The empirical evaluation reported
in Chapter 6 suggests that this was a mistake.

3.3.6 Communication with External Interfaces

Standard software engineering methodologies recognize that any program or
system exists within a larger context [22]. Few systems solve a problem without
communicating the solution to a human or other automated agent. Most systems
gather information about the world beyond their own knowledge bases to compute a
solution to a specific problem instance. The theory and tool developed in this thesis
must account for input from and output to the world beyond the problem being
solved.

This is the role of leaf nodes and the goal node on the conceptual map. SITE
made use of that fact implicitly by attaching knowledge frames to leaf nodes. The
Computational Theory of Design includes an explicit description of leaf nodes as
knowledge sources, but lacks a corresponding description of the goal node as a logical
output of a final value. Logically, concepts with no supporters must obtain their
values either from built in knowledge bases or by interaction with the outside world.
Designer constrains leaf nodes to be annotated with constraints to accomplish one of these.

In practice, information can be output at any concept, by means of the usual Prolog side effects. User input may be required at higher levels also, as a response to other knowledge already collected from the outside. Designer addresses these issues pragmatically. A more complete theoretical treatment is needed than was achieved in this thesis.

3.3.7 Metaknowledge

A significant issue involved how much domain knowledge to build into a model editor system, and for which domains. In contrast to automatic programming systems [11], this system avoids detailed metaknowledge of both specific problem domains for which designs are to be developed.

Many AI programs use metaknowledge at different levels of abstraction to build layers of control governing the tool [37]. The original investigation of a system to regulate its own updates expected to use several control levels. When the metaphor of editing a conceptual map was adopted instead, there was no need for layered metaknowledge. Most control in Designer is abstracted into a small number of control patterns. Each of these operates on a single layer of data and is inherited by most of the classes that descend from the conceptual map class.

The theoretical description in Chapter 4 of design activities was realized as directly coded procedures, that being a more direct way of obtaining a prototype program. An alternate architecture could have expressed this theory declaratively,
and built a control structure to execute it. This second choice might more easily allow changes in which design activities are automatically executed.

Metaknowledge is used for the control constructs in the Human-Computer Interface (HCI). Here, a standard set of frames represent the semantic content of messages between modules of Designer and external agents (either the user or other modules). The same control objects operate on any interface style, for example a text interface, and use metaknowledge to route syntactic events to objects that implement the specific interface currently in use.

More significant metaknowledge was used in knowledge bases that describe design constraints and editing commands. These are described more fully in Chapter 4. The code generation knowledge base contains a set of rules that operate on concept nodes to produce Prolog code. This can be considered metaknowledge, in the sense that it encodes knowledge about the language. However, it is more like procedural operations than like declarative descriptions of Prolog.

3.3.8 Code Generation

Code generation was an assumed initial goal, though the iterative editing of a partial design was assumed to be much more important. Since the program SITE had successfully generated Prolog source code, it was assumed that it would be trivial for Designer to do the same.

Generating code became more significant after the theoretical conclusion was reached that code generation is not different in type, only in detail, from refining a
design in higher levels of the conceptual map. It was then assumed that the system would demonstrate this claim by generating code.

The choice of Prolog as target language was also taken for granted before the project began. However, there are good reasons for this choice. Prolog is a close match to the computational model and data structures used in the annotated conceptual map. It is syntactically simple. Both features reduce the amount of knowledge about Prolog required in a code generator. Designer automatically transforms the declarative knowledge stored in a conceptual map to Prolog constructs. Very little additional knowledge is required of Prolog's unique language features.

3.3.9 Applicability to Actual Software Projects

Section 3.4 explains the real-world setting in which this thesis is to be applied. For the Designer tool to be transferred to real software development projects, it must support more than sample problems. This requirement reveals an assumption that conceptual maps will scale well to problems that contain tens of thousands of concepts.

Scaling up from sample problems to large, real environments is often a problem for AI projects (this problem is discussed, for example, in [36] and [19]). This was the main reason for introducing multiple levels of refinement in conceptual maps. There is no theoretical need or advantage for them, but practical experience in software engineering [22] gave good reason to assume that a single-level map would break down if used for a large system.
3.3.10 Intended Audience

A final assumption is the intended audience. Any person with a background in software engineering methodologies should find the underlying concepts familiar. The theory in Chapter 4 is applicable to any technical or engineering discipline. To read and use the formalization of the theory requires a background in basic graph theory, logic and constraint expressions, and common knowledge representation techniques.

The tool that was built from the theory is intended to be usable by any professional with an understanding of the target problem domain. Although Designer attempts to introduce modelling concepts such as choice and iteration intuitively, some prior experience with process models or software methodologies would be helpful. The experimental results reported in Chapter 6 indicate that experience with the theoretical background is not needed to use conceptual maps successfully.

3.4 PROBLEM CONTEXT

This thesis intends to give practical help to designers in a real software maintenance environment. The target application domain is AEGIS, a large and complex naval shipboard real-time control system. This system is very well-defined: existing specification documents are exhaustive, and very close to the implementation level. AEGIS is implemented in a very low-level procedural language, CMS-2. It is supported by an extensive maintenance organization.

AEGIS design documents suffer the disadvantages discussed in section 1.1, and are seldom used by maintainers. The system is expensive and difficult to
maintain. A conceptual map that is very close to the existing official specification of
the system can be developed for AEGIS. The goals of supporting design, analyzing
alternate design choices, maintaining a design for later reuse, and analyzing software
problems at the design level are all expected to be achievable for AEGIS.

However, the code generated by Designer is not immediately applicable to this
environment. The Prolog code generated by Designer is very different from the
CMS-2 source code used in AEGIS. Chapter 4 discusses the prospects of future
enhancements in code generation fitting these needs better.

AEGIS imposes very specific timing and memory constraints on the executing
program. The Computational Theory of Design does not consider run-time
characteristics of the systems that implement designs, and Designer contains no
knowledge about required or actual performance. It would not be hard to add this
knowledge as additional annotations to a conceptual map. To generate code that
makes use of run-time knowledge, or could guarantee that it met run-time constraints,
would require techniques from the real-time literature that are far outside the goals
considered in this thesis.

3.5 ON SELF-REFERENTIAL PROJECTS

This report advocates a consistent refinement of software products from
concept to code. One result of this thesis is a software product. If the approach
described herein is worthwhile, then this thesis itself should use these techniques to
refine the theoretical concepts to code.
Since the design support tool developed during this thesis was not available to bootstrap its own development, the thesis used a hypertext system to manage the project. The nodes of this hyperdocument contain report text, design documents, User Action Notation [14], several types of graphics, and Prolog source code. There are several intermediate levels within the design. Each level is connected to the others by hypertext links.
CHAPTER 4
A COMPUTATIONAL THEORY OF DESIGN

This chapter describes the Computational Theory of Design. An informal description is followed by a formalization of the theory. A proof is also presented of its computational generality.

The literature from several different problem domains and disciplines develops assumptions and descriptions of design that have much in common. The design of software systems has more similarities than differences with the design of other artifacts. The theory of design described here is applicable to any design problem. The Designer tool supports conceptual maps that would be as useful to architects or mechanical engineers as to software designers. The problem solutions it generates from a conceptual map, however, are specific to software design problems. The primitive computations in its constraint annotations, and the Prolog source code generator, are unlikely to be useful to architects.

4.1 META-THEORY: PROPERTIES OF A THEORY OF DESIGN

Section 3.3 identified several assumptions about background requirements and desired characteristics for a design theory (and the resulting tool). These assumptions are summarized as follows:

a. Emulating a human activity in an automated program is an effective way to automate the activity.
b. To emulate a human activity, a program must have some understanding of the activity.

c. Knowledge about an activity can only be automated if the activity is well-understood. Equivalently, there must be a relatively detailed theory that describes the activity.

d. Formalization of a theory is desirable. A formal theory can be implemented in a computer program to perform the actions described by the theory.

e. A model theory is a useful style to use for a theory of design.

f. Prolog is a useful computational model for a theory of design.

g. A declarative descriptive style is preferred through the theory and implementation. Declarative knowledge is more easily expressed, understood, maintained by humans, and can be manipulated by simpler control structures.

h. A theory of design should be sufficiently formal to be expressed as logic statements. An intuitive interpretation of each element of its logic statements is a sufficiently detailed level of formality. Such a theory will translate easily to Prolog.

i. It is best to keep design work at a high level of abstraction for as long as possible. This "highest level" is identified with work on the problem setting. Therefore, the fundamental activity of design (and the
center of a theory of design) is to represent and manipulate a changing problem setting. The manipulations include the eventual transformation of a problem setting to a problem solution.

j. A single theory and associated knowledge representation are desired, that are general enough to use for any design problem in any arbitrary problem domain. A sufficiently general knowledge representation is likely to contain a simple syntax and semantics.

k. The conceptual map is a sufficiently powerful representation.

l. Conceptual maps will scale well to large problems. However, a single-level map will not handle large problems. Additional hierarchical modularization will be needed.

This list leads to the following, more specific list of desirable properties of a theory of design:

m. The theory must represent what human designers do, and how they generate, choose, refine, and edit the individual elements that make up a design.

n. Humans do not generally do things in a formal manner; therefore, the theory must represent informal human processes.

o. While having no specific knowledge of the contents of any particular design, the theory must represent the structure common to all designs. It must represent arbitrary components (data and processes),
structure, and other relationships between components that may be used in specific designs. The theory must represent and manipulate communications between the problem setting and interfaces outside the immediate problem context. It must know the operations on the components of design, and relate these operations to the human activities described above. It must know which operations can be automated, and how to complete them. It must represent constraints on the components of a design, and know under what circumstances the constraints apply. It must include rules about how to manipulate design components, and rules about how to make inferences from the content of a design.

p. The theory must include a general computational model. It must know the meaning of computing a value for a design, and know how to connect its computational knowledge to the components of a design.

q. The theory should be simple and robust.

r. The theory should use a logical form compatible with, and easily translatable to, an executable target language (the specific target chosen in this thesis is Prolog).

4.2 AN INFORMAL THEORY OF DESIGN

This section presents an intuitive and informal description of design. The theory is formalized in section 4.4. A model is developed of the actions performed by human designers. The model does not attempt to make decisions about design
choices, but to represent the results of those choices, the contents of a design, and the operations applied to it. This model approach attempts to mimic, and exploit, these basic human behaviors. Design is taken to be a paradigm of human problem solving.

In overview, a designer begins to describe a problem by identifying the concepts involved in the problem and its setting. Design then enters an open-ended phase of editing the problem setting. The conceptual map is used to support this period of exploring and discovering the relationships among the concepts affecting the problem. Editing includes both extensions and refinements to a conceptual map.

A designer does not necessarily approach a problem with a desired solution in mind. Part of the purpose of exploring the problem is to help the designer discover or invent the concepts that represent desired solution states. When the problem is sufficiently well-understood and -described, it includes concepts that represent solution goals.

The designer introduces a problem solution to the design by constraining the concepts that support the solution goals. The problem setting has no computed value. When a part of the design is constrained in enough detail that its value can be computed, a part of a solution has been created.

When the entire conceptual map has been annotated to constrain the values computed by each concept, it can be executed to compute a value for the solution goals. This value is a problem solution. In practice, the map is executed by generating source code for a computer program, which can then be executed.

This theory is called the Computational Theory of Design. Computational refers to the ability of a fully annotated conceptual map to compute a value for each
concept in the map. Values are combined according to the constraints among the concepts to compute a final problem solution.

4.2.1 Problem Setting and Problem Solving

This section establishes basic definitions of problem description and problem solution. Although there is a close relationship between them, they are distinct and recognizable. The first part of design is to discover what the problem space actually is. Problem solving begins when the designer bounds the setting, and begins to add enough constraints to define a particular problem solution, distinguishable from all others.

Many AI programs, including the first generation of heuristic classification expert systems, depend on search through a problem space. A search for a solution succeeds if the solution to the problem is already encoded in some way in the search space. The problem description must include the problem solution.

If design were a search through a solution space, as in Simon, then design would have to construct a space of sufficient possible solutions. But in design, the product of the design task is something new: a design is not present in the database with which a designer begins. This theory follows Eden and Schön, who describe a process of constructing first a problem setting, and then a problem solution.

If a problem solution were contained in the problem setting, then an appropriate theory could generate a problem solution from a very early stage in problem analysis. Schön's empirical evidence suggests that humans may begin looking for the problem solution very early. Eden and Simon observe independently
that appropriate solutions often become obvious in a well-stated problem. Designers may use the problem description as a default first draft for a problem solution. However, solutions do not fall out of descriptions automatically.

There is some tension between the claims that there is no essential difference between design and implementation, and that problem setting must remain distinct from problem solution. It is very useful to recognize the crossover point from problem setting to problem solution, because it indicates the point at which a design contains enough knowledge to generate values that satisfy the problem goals.

The difference between problem setting and problem solution is that a problem solution contains enough information to compute a result. Early in its life, a design contains partial descriptions of constraints that have not yet been fully defined. They constrain only the structure of the design, not its value. This is the nature of problem setting. Later, when constraints are fully instantiated, they describe a problem solution. If a concept is an abstraction for some extensional set of things in the problem domain, then a problem solution constrains each concept to represent one value of its set. A fully-constrained design can be transformed to an executable program that generates actual solution values.

The Computational Theory of Design represents both problem and solution with the conceptual map. A single representational structure is used here for both problem description and problem solution. Keeping the two phases in a single model of design activities is more amenable to automated support than is the traditional taxonomy of requirements, analysis, design, and code. When used as a problem setting, the conceptual map is a very general tool for viewing, editing, analyzing, and experimenting with a problem description. When instantiated as a problem solution, the map computes a value for the design.
Different writers talk about problem "environments," "domains," and "settings." The following definitions are useful here:

a. Problem Domain.

A domain is a general group of concepts, involving a single subject area. A problem domain is the set of objects and processes that a designer must know about to talk about the problem.

b. Problem Environment.

An environment includes concepts beyond the specific domain of a problem. To understand a problem, a designer must consider the problem domain within other related domains.

c. Problem Setting.

"Setting" is used in the sense of setting up a problem so that it can be manipulated. A problem setting is a description of the problem. It includes as much of the problem domain and problem environment as is needed for understanding. "Problem setting" is synonymous with "problem description."

d. Problem Solution.

A solution is some artifact or process that changes the state of a problem environment to one that is no longer considered to contain a
problem. It is sometimes used to mean a description of, or design for, the actual solution.

4.2.2 The Process of Design

If design is the process of making a plan to move from an existing environmental state to a preferable state, then something in the existing environmental state can be characterized as less desirable than the imagined, or hoped-for, future state. This characteristic of being less desirable is commonly called a problem. Design must first understand and describe the problem, then describe some solution to the problem, and finally create a plan to create some artifact that can solve the problem.

Design is described in this theory as a broader activity than planning. The two can be defined in alternate ways that make them seem almost alike. Although many writers define design as the process of planning an artifact, planning systems commonly emphasize the problem solution domain. Many such systems represent a solution space, and develop steps to arrive at a goal in that space. Planning is a knowledge activity concerned with how to control the process of creating new plan steps.

Design emphasizes the problem space. Understanding the problem setting is a significant activity, from which the planning steps may work forward before a solution goal is fully defined.

The close relationship of design and planning is seen in Dasgupta, who uses the image of a plan very explicitly to explain the process of design. A plan contains
declarative instructions for how to construct the artifact. These instructions are analogous to the annotations in a fully constrained concept map. A design also contains models of the problem setting and, eventually, of the desired artifact. Both are at a more abstract level than the actual construction of the artifact. Because of this, designers can easily manipulate and reflect on the plan. It is the finished artifact, when constructed according to the plan/design, that causes some change in the human environment.

The Computational Theory of Design attempts to take advantage of what humans are observed doing while solving problems. Eden and Schöen's observations are informal but empirical, and cover a broad range of design domains. They report that the actual practice of designers does not follow the current software engineering model of analysis → requirements → solution.

The identification of a problem is completely subjective. A problem exists only as a human like or dislike of the state of an environment. What identifies a state (e.g., too little water for a community) as a problem is not the quantity of water, but that the community regards it as insufficient. The subjective nature of problem identification implies that a requirements specification will capture one set of requirements, as they appear to one group of humans. Because humans capture only a part of the entire problem setting, no requirements specification will be complete.

This implies that the traditional requirements specification is not adequate for understanding a problem setting. Although simple problems may produce feasible requirements identification, the same methodology breaks down with large problems. After the requirements are captured, practitioners do not rationally map the problem onto a solution. The problem of gaining customer agreement on a requirements specification is well known in traditional software engineering (see DeMarco [6]).
is also well-known that requirements are not stable, and that the human view of the problem changes over the life of the project. This implies that a requirements specification established at the beginning of design may not remain valid throughout the design process — a prediction borne out by the literature (see [27]).

Rather, designers iteratively make small changes in a model of the problem setting. Constructing a model of the problem environment captures it in a representation usable by humans. As a problem is often actually a complex of problems, there are many possible problem settings. The beginning of design is the analysis of the problem and its context. The designer will edit, propose, add, and think about the problem setting itself, whether or not solutions can yet be seen in the setting. This is Schön’s act/reflect cycle [34]. The early stages of messing about with problems (Eden’s phrase) are iterative and open-ended.

A designer may have some specific goals in mind, corresponding to some nodes that will eventually exist in the finished design graph. However, the theory does not require any initial goal at all. Intuitively, it is very possible for a designer to recognize a problem, but have no initial idea of how to solve the problem. The theory permits working with the problem setting alone for an indefinite time. Solution goals will be identified incrementally as the problem is better understood. Commonly, a design in progress will contain a partially defined problem description and a less well defined set of high-level goals.

Humans do not represent or solve a large problem all at once. They work at it iteratively, in the way that one might tease loose ends out of a tangle of yarn. A design theory must support that behavior. In Dasgupta’s interpretation, the process of design is constrained by bounded rationality. A human mind can only work on some small chunk (in the sense defined by Miller [23]) at a time. The designer can
therefore find, browse, and change some small subgraph of the conceptual map. The map is neither monotonic nor stable.

The map is a dynamic model of an ever-changing design. The problem setting eventually evolves into a problem solution. This suggests Simon’s model of transforming a problem statement into a solution [39]. For Simon, design finds a way to represent the problem that makes the solution obvious.

The next sections describe the objects manipulated in the design process, and the operations permitted in the Computational Theory of Design to accomplish this transformation of problem to solution.

4.2.3 Objects of a Theory of Design

In software development, all of the raw materials for the design and construction of programs are knowledge. The objects of this domain are ideas and actions. Data structures and process descriptions are mental constructs. Software development is thus a specialization of knowledge manipulation.

The conceptual map, as already introduced in section 1.4, provides a way to represent, communicate about, store, and explore a design. Besides the conceptual map itself, a design may deal with several higher-level features. The abstract objects of the Computational Theory of Design are the following:

a. the conceptual map,

b. concepts,

c. support relationships among concepts,
d. values of concepts,
e. constraints on concepts,
f. choices, and
g. loops.

The structural relationship among these features is shown in Figure 4. In the diagram, outer boxes enclose their component parts. Several features have additional relationships between each other. This section discusses their intuitive meanings. See also Figure 10.

Design in any discipline uses mental constructs to represent objects and actions in its problem domains. The easiest model to manipulate is a mental model. Generic design is described by a general theory in which the objects of the theory are concepts.

Figure 4: Structural Decomposition of Conceptual Map Components
A problem, or in Eden's words a problem mess, is a tangled collection of concepts that influence each other. A problem can be represented as a map of concepts. The concepts are the primitive elements of design activities. The work of design is to refine and understand the relationships among these concepts, and eventually to build a version of the conceptual map that represents a solution to the problem.

The concepts of the map are completely general. They can represent data, as in the example of section 4.3.1, processes, as in the example of section 2.1.1, or more abstract concepts, as in the example of section 4.3.2. The graphical notation described in Eden is used to depict conceptual maps informally. See Figure 1 for an example.

A collection of concepts is given structure by the support relationships between them. In Eden, a conceptual map is a potentially large, two dimensional space, very much like a conventional road map. Each concept is directly connected to each of its influencers.

The "Structured Software" revolution [22] has given good evidence that large systems should not be represented in a single level of detail. Concepts are more typically broken into a modular hierarchy, recursively breaking down abstract concepts to more detailed concepts.

The Computational Theory of Design supports this by representing a conceptual map as a three-dimensional structure, similar in some ways to Stefkï’s problem spaces [42]. In a horizontal dimension, a concept is supported by other concepts at approximately the same level of abstraction. In a vertical dimension, a concept is refined to a set of more detailed concepts.
It is possible for a conceptual map to include horizontal subgraphs at different levels of detail, each refining different parents in the vertical dimension. Thus, returning to the example of section 1.4, as illustrated in Figure 1, an "operator" requires the conceptual support of a "training mode", but the meaning of the concept "training mode" is refined to a lower level of detail. A "training mode" is defined to be exactly one value from a set of possible values. Each of the other concepts in the example will probably, in a complete design, be similarly refined to a set of primitive-level concepts.

The simple map captures the structure of the design. Though the claim of many writers that structure is the essence of design [5] seems sound, the structure alone does not carry sufficient knowledge to generate a problem solution. If algorithm = logic + control, then design = structure + algorithm. The structure of a conceptual graph encodes the control decisions and logical steps of the algorithm. Annotations to the concepts record constraints between them, which in turn define the content of each step.

During the design process, higher-level nodes are annotated with high-level constraints. These are refined to more detailed constraints as the map is refined to contain lower-level nodes. The supporting relationship says there is some dependency between concepts. The constraint says what it actually is. A constraint attached to one concept may also use several supporting nodes, effectively combine the binary support relationships into a multiple-part constraint.

Design is a computational activity. A computation creates a value. The purpose of the constraints is to determine the value for each concept. Eventually, constraints are specific enough to define values for concepts. Each concept is said to have a value. Each concept then computes its value from the values of its supporting
nodes; the value of each concept supports the computation of the ultimate value of the entire problem solution.

If a conceptual map contains different levels of detail and refinement, then each level computes a result at a different level of abstraction. The higher levels of a conceptual map are suitable for high-level human discussion and reflection on the design. They are refined into middle levels, which express explicit computations. Problem solution constraints can be applied at these middle levels. Levels below these middle levels represent primitive facts and values that cannot be computed, but are obtained from outside the problem.

A concept may have a simple value, or an arbitrarily complex one. The value of a concept can represent a primitive data value, a complex data structure, or a process.

Eden associates a concept with a pair of values, representing a range. He labels each supporting arc as having a positive or negative influence on the supported concept. Eden takes this feature from System Dynamics. This is an abstract and informal definition of a data type for each concept in a problem setting. The Computational Theory of Design is more interested in the specific value to be computed for each concept when the problem solution is executed. Although there are interesting and useful inferences that can be made from types, the current theory does not have a specific need to develop a type system on a conceptual map.

Type is related to the notion that a concept represents a set of values. Before the computation defined by the design is executed, it is an abstract representation of the set of values that the concepts may take during different executions. The design is a structure in which a solution is realizable. An actual solution is not defined until
the artifact resulting from the design is executed to solve an actual problem by computing an actual result.

These computations must be described in the design, which is used as a plan to create an artifact that executes the computation to create values. Because the theory described here includes the automatic generation of source code, building and executing the artifact is equivalent to executing the design.

Intuitively, humans devise a procedure to describe how to perform a computation. The Computational Theory of Design assigns an explicit procedure to each concept, describing how to compute a value from the concept’s supporters. It computes a more specific value for a node than Eden’s original work. It is a more static approach than System Dynamics, in which interacting constraints are iteratively satisfied until the network comes to a stable state.

Physically, each procedure is represented as a declarative constraint. The purpose is to keep the designer’s attention on a statement of what the relationship is among a set of concepts. This helps keep the design work at a more abstract level, rather than focussing on the lower-level details that are required by most procedural languages.

Annotation is a generic device. A concept could be annotated with many types of knowledge besides the constraint on its value. For example, Dasgupta annotates his design constraints with the history of design decisions that justify a current constraint. If type information were desired, it could be easily annotated to the concepts on the map. The Computational Theory of Design uses a small amount of other knowledge about concepts. These additional annotations are used for bookkeeping, and are described in section 4.4. The term "annotation" is generally
used here synonymously with the constraint annotated to each node to describe how to compute its value. The infrequent uses of other types of annotations will always explicitly make clear that they differ from the constraints.

Constraints are not shown in the graphical notation for conceptual maps. A constraint may be indefinitely complex. It would be difficult to depict large constraints on even a simple map. It would also be difficult to depict even simple constraints on a large map. This difficulty is analogous to that found in Entity-Relationship-Attribute (ERA) modelling. The diagrams of that technique become difficult to read when they contain even modest numbers of attributes. It has become much more common to depict only an Entity-Relationship Diagram (ERD), with attribute information recorded in a separate text document.

Similarly, constraints are recorded and used as text associated with the graphical record of a map. In this document, an informal notation based on Prolog is used to discuss constraints. The notation used in the tool Designer is discussed further in section 5.8.

A constraint does not always describe a simple computation that combines all of its supporting concepts. The control construct of choice is represented in a constraint that describes more than one way to use its concept's supporting concepts.

Choice is found in even the simplest computations. The simplest example is the logical statement:

\[
a \lor b \text{ implies } c.
\]
This statement is modelled in the conceptual map of Figure 5. This shows only the structure of the statement. Its meaning so far is merely that c is supported by both a and b. The additional semantics of choice is found in the constraint that expresses what the relationship actually is among a, b, and c.

The logical statement above is a conditional expression. In a traditional programming language, choice is expressed as an action that results if a conditional expression is true:

```java
if (a or b)
    then c;
```

For use in a declarative constraint style, this is recast as

```java
compute-from(c, a, b) if
    true(a) or
    true(b).
```

Physically, it is convenient to consider the different cases in a choice as refining concepts of the parent choice. This makes clear that the notion of a choice is a higher-level feature than the concept that makes the choice.

This becomes clearer when more complex conditional expressions are used. The above example is a trivial case: if either a or b is true, then c is true. Most
computations make choices by testing the truth of more complex conditional expressions. An example constraint that is slightly more general is:

\[
\text{compute-from}(c, a) \text{ if } \\
\text{greater-than}(a, 0), \text{ or } \\
\text{compute-from}(c, b) \text{ if } \\
\text{greater-than}(b, 0).
\]

The constraint in this example has two independent clauses, only one of which is used to compute the value of the concept. The conditional expressions serve as guards (using the term from Hoare [15]) to determine which clause is used to constrain the actual value of the concept.

Guarded constraints are very general. In the example above, the relationship defined by the constraint is trivial if the clause is true. It is also common to define a constraint that holds whenever a guard is trivially true, that is, always. And-nodes that are always to be executed have trivially true guard expressions.

The guards are used for convenience, not by necessity. Constraints are closely related to simple Horn clauses. A Hoare guard in a Horn clause produces what might be called a "guarded Horn clause". This structure is logically redundant. Any subpart of the constraint's definition can function as a conditional expression (the sequential semantics used here would require the subpart to be a sequential set of
subgoals beginning with the first) [20]. However, identifying an explicit conditional expression in a constraint makes the declarative semantics resemble the familiar, intuitive "if-then" expression of a choice. Later, in Chapter 5, it will also regularize (and thus simplify) the implementation of both choice and looping.

\[
\text{compute-from}(c, a, b) \text{ if }
\]
\[
\text{true,}
\]
\[
\text{add(a, b, c).}
\]

A choice constraint may also choose from more than two possibilities. The theory assumes that the constraint guards are evaluated in sequence, and applies the first constraint for which the guard is true. This follows the computational model of Prolog [20], so that any combination of choices can be expressed with this control mechanism.

Most problem settings also include loops. These may represent feedback loops found in the real world, as in the business example of Figure 2. They may also represent computational iteration over a set of objects, as in Figure 6 or a recursive definition of a computation.

These examples illustrate how naturally a conceptual map represents loops. In System Dynamics, and frequently in simulators, loops are dynamic models. An executing model iterates through the loop, usually updating values until a steady state is identified.

The Computational Theory of Design uses loops as deterministic calculations. Although the simulator dynamic model could be represented in the theory, such an extension is currently left up to the user. All loops in a conceptual map compute a
Figure 6: Map of Iteration through a Set

single value, although the computation may be recursive.

Intuitively, a loop is a control construct at a higher level than the concepts and the cycle of links that are the components of the loop. In traditional programming languages, the semantics of a loop is to reuse the component calculation steps within the loop to compute a final value for some set of concepts. This is true both for a procedural loop construct, for example:

\[
\text{for (i = 0; array[i]++; i++)}
\]

and for more declarative recursive styles, such as:

\[
\text{increment([], []).}
\]
\[
\text{increment([Head0 | Tail0], [Head | Tail]) :-}
\]
\[
\text{Head is Head0 + 1, increment(Tail0, Tail).}
\]
Both traditional constructs use a header structure that makes visible the data structures updated by the loop. The Computational Theory of Design makes this structure explicit. Each loop in a conceptual map identifies its entry point. This is a distinguished concept that serves as a controller.

A loop entry point, or header concept, serves two functions. First, any values computed by the loop that are to be made available to other parts of the conceptual map are found in the value of the header concept. As with other values, this may be used to package a set of values if the intent of the loop is to generate more than a simple value.

Second, the header concept determines the exit condition of the loop. Every loop must know when to end. As with a recursive function, a loop in a conceptual map begins at its header concept, which tests whether its exit condition is satisfied. If not, the header concept uses the values of its supporting nodes to compute a new value. Computing the values of a supporting node in a loop leads eventually back to the header concept. The loop continues until the exit condition is satisfied. At this point, the header concept has computed a value, which is used iteratively to construct final values for each cycle that was instantiated through the loop.

A loop uses the same underlying control construct as does a choice in the conceptual map. The header concept is annotated with a constraint that can be satisfied by more than one clause, depending on the truth values of conditional expressions, or guards, associated with each clause.

Figure 7 shows the structure of a loop.
Figure 7: Structure and Knowledge Associated with a Loop

The structure of a conceptual map, with constraints on each node, could be interpreted within more than one computational model. Intuitively, it does not matter how values are computed as long as each computation has access to the values of its supporting nodes.

If a parallel implementation is assumed, this would be a natural dataflow model. However, this thesis assumes that the theory will be realized on a sequential computation device. Therefore, the discussions here view computation as goal-driven and backward-chaining. This is, of course, how the Prolog language computes. For each concept to achieve the goal of computing its own value, it must first trigger its supporting concepts to compute their values. The supporting concepts play the role of subgoals.
4.2.4 Operations on the Objects of the Theory

Editing is chosen as a central metaphor to model the fundamental activities of design. As explained in section 2.1.3, it is a very close fit to Schön’s description of the iterative cycle of design activities. It also satisfies Shafer’s requirement, discussed in section 2.2, for sufficiently rich modelling elements to avoid model aliasing. It implements Eden’s description of incremental changes to a partial conceptual map.

Editing is an appropriately large and rich notion. It is also well-understood, and simple to describe: every edit is a combination of additions to and deletions from the primitive elements of the knowledge structure being modelled. If that knowledge structure is a rich enough representation of the problem domain being modelled, then a model that provides general editing on the knowledge structure is a complete model. When the primitive elements include both entities and relationships (as when the elements include concepts and support relationships between concepts), a simple edit model can transform both the structure of the conceptual map and the individual concepts in the map.

If the edit operations are additions and deletions, then the operands of those operations are concepts, individual support links, constraint annotations. At a higher level, choices, loops, and the entire conceptual map can act as operands of edit operations. These higher-level features will usually be edited by editing their constituent parts, rather than by deleting the entire feature.

The effect of an edit operation is to transform one version of a partial map into another. Since a concept can also be refined to lower-level concepts, editing a concept may also mean adding its refining concepts, which implies adding the
refinement relationship between them. There are many possible levels of abstraction between the top and bottom layers of a conceptual graph. Refinement may involve restructuring the nodes at some given layer, or moving nodes between layers of different levels of abstraction.

Refinement may also make constraints on concepts more specific. A high-level conceptual map of a problem setting can be refined toward any one of a family of possible problem solutions. In Simon's terms, the map describes many partial paths toward a solution. Refining the constraints on the map prunes many possible solutions, until the conceptual map is constrained to represent only one possible design. Each tightening of a constraint represents a decision that some of the available partial paths are unlikely to lead to acceptable solutions. This is also congruent with Barstow's work on refinement as a paradigm of automatic programming [11].

At the lowest, most detailed level, a constraint must describe the creation of data. The primitive data used in the lowest-level computations comes from somewhere. The types of objects that can be added by editing the lowest-level concepts include primitive operations to generate simple data values, to construct and access constant knowledge bases, and to obtain data by interacting with the environment beyond the conceptual map. Pragmatically, other operations to perform common low-level computations (such as arithmetic) are included in these primitives.

The lowest levels of the conceptual map are eventually refined into a programming language. This step is completely automated. Code generation is a special case of refining a design to lower levels of detail. It is also a special case of computation on design.
The paradigm of a conceptual map generating computational values emphasizes process knowledge. Software engineering practice uses multiple views to build an adequate understanding of a problem setting. Concepts can represent other types of knowledge as shown in the examples of section 4.3. A strength of the Computational Theory of Design is to carry all of this knowledge — conceptual, declarative, and procedural — in a single representation. This is analogous to the use of First Order Logic as a unifying paradigm (see Genesereth & Nillson [20]).

4.3 EXAMPLES

These examples are drawn from the project's actual problem domain, the AEGIS software system. They contain no actual AEGIS values. They are based on unclassified portions of the AEGIS Command and Decision (C&D) Program Performance Specification (PPS), approved by NSWCDD/N86 for use as examples.

4.3.1 Example: Data Structure Model

Conceptual maps can represent both low-level data and abstract knowledge about data. Figure 8 shows a traditional ERA data model of the top-level relationships in the AEGIS software. It is instantiated as many specific software entities and relationships.
Figure 8: ERA Diagram of Part of the AEGIS Data Model

A Computational Theory of Design
A conceptual map similar to that shown in Figure 9 could be constructed from Figure 8. A similar map could also be constructed using concepts to represent specific instances of element, baseline, modules, sysprocs, and procedures.

Figure 9: Modelling Data Structure in a Conceptual Map
Some information is lost in this translation. The relationship links of ERA represent a variety of relationships, all with distinct semantics. All these are mapped onto the conceptual map's support relationship, with its very generic semantics. To use the knowledge of the original ERA representation, it must be captured in the constraints annotated to the entity concepts.

4.3.2 Example: AEGIS Command and Decision

The example introduced in section 1.4 (see Figure 1) can now be refined. The leaf nodes in Figure 1 would, in a complete map, be refined to lower-level concepts. The concept "doctrine option" would be constrained to represent a constant value, such as:

```
compute-from(doctrine-option) if
    construct-data("template", doctrine-option).
```

where "construct-data" is a primitive operation that binds the value of "doctrine option" to a data structure that is explicitly specified as a constant value.

The concept "operator" would be refined to make use of interaction with the end user of the system, such as:

```
compute-from(operator, training/tactical) if
    value-of(training/tactical) and
    get-interactive-selection-from(training/tactical, operator).
```
where the operations are primitive operations that access a supporting concept value (in this case, a list) and ask an interactive user to select an item from that list.

The concept "track" is similar to "operator". The higher-level concepts will all use data constructor primitives. The top-level concept "doctrine statement record" could use a primitive operation to output its value to a user. It is more likely to support other concepts in a larger map of a more complete system. The figure depicts only a small subset of C&D. A fully-expanded version would be a hierarchy of many conceptual maps, connecting several levels of abstraction.

4.3.3 Example: Meta Representation

A common test in Computer Science is whether a system includes the expressive power to represent itself. Figure 10 is a conceptual map of the Computational Theory of Design.

This thesis currently takes no advantage of this ability. However, it could support future extensions to the specific design constraints used in the implementation of the theory. Figure 10 is a declarative representation of the meta-knowledge about design. Section 8.9 discusses this possibility.

4.4 FORMALIZATION OF THE THEORY

Section 4.2 described the basic activities and operations of design, and the objects in the domain of those operations. The intuitive description is formalized here. The formalization describes a model-based system for representing a problem.
and solution. The model is a declarative analog of a designer’s intuitive description of a problem setting.

The formalization is the basis for Designer, a tool that implements the theory as an automated design assistant. It produces an executable version of the design model by automatically generating Prolog source code.

The Computational Theory of Design describes how humans do design. It follows Simon’s prescription [39] that such a theory should describe the design and the design process. However, much of the human process of doing design work is informal. Some of the informal activities described cannot be modelled in great detail. Those informal activities cannot be automated, but can be described in the formalization and supported with automated tools.

The conceptual map described in section 4.2.3 is represented as a general graph, whose nodes represent concepts, and whose arcs represent support.
relationships between concepts. This structure is called here a conceptual graph, and its nodes as concept nodes. Because the map can be refined to different levels of detail, arcs of the graph may be either support arcs or refinement arcs. The conceptual graph is actually a network of subgraphs, hierarchically related by refinement arcs.

The constraints specifying the value of each concept are annotated as attributes of the concept nodes. Each constraint is described with a structure and notation discussed in section 4.4.2.

Choices and loops are not explicitly represented as high-level objects in the theory. The presence of each can be inferred from the structure of the graph and the presence of conjunctive or disjunctive clauses in the constraints on nodes.

4.4.1 Interpretations and Analogies with Other Representations

The conceptual map is closer to the augmented transition network, as used in semantic theory or formal language theory [29], than to a simple graph. There is also a close analogy between a conceptual graph and a syntax tree. The syntax tree shows sentence structure, just as a conceptual graph shows problem structure. According to formal language theory [29], a syntax tree is only powerful enough to express a context free grammar. Semantic language features that cannot be expressed within a context free grammar are represented as attributes to the grammar, and by corresponding attributes attached to nodes of a syntax tree.

The conceptual graph is intentionally similar to the search tree of a Prolog program. In Prolog, the sequence of goal satisfactions that describes the execution of
a logic program has the basic structure of an and/or tree. Prolog gains its
Turing-equivalent expressiveness by supporting communication of variables, which
represent context, between branches of its tree.

The conceptual graph shares this structure. As with Prolog, it permits cycles
in the nodes of the and/or tree. It achieves an analogous Turing-expressiveness by
making values from computation subtrees available via computed values of nodes.

4.4.2 Structure and Components of the Model

Let Graph = (Nodes, Arcs) be a directed, possibly cyclic, graph. The
graph represents a conceptual map. The nodes represent concepts. The arcs
represent relationships between concepts. As with all graphs, Arcs is a relation ⇒
mapping Nodes onto Nodes.

Nodes = \{n_1, \ldots, n_n\}

Arcs = ⇒: Nodes → Nodes

= \{n_i ⇒ n_j, \ldots, n_m ⇒ n_n\}

The relationship between concept nodes has a deeper semantics than in Eden.
An additional semantic is introduced by the higher dimension of the map: arcs in the
graph either support nodes at the same level, or refine a node from a higher level (see
Figure 3). Thus, there are two types of arcs connecting nodes n ∈ Nodes:

Arcs = Supports ∪ Refines, where

Supports = \{n_i ⇒_s n_j, \ldots, n_k ⇒_s n_l\}

Refines = \{n_i ⇒_r n_j, \ldots, n_k ⇒_r n_l\}
The concept support relationship \textbf{Supports} represents the set of specific influences between concepts. Following Eden, no intrinsic meaning is attached to a conceptual support arc. Informally, $n_i \Rightarrow_s n_j$ suggests that $n_i$ influences $n_j$ in some way, so that $n_i$ depends on $n_j$. Pragmatically, more specific interpretations may be attached to this relation during computations on the graph. The refinement relationship \textbf{Refines} represents the refinement of a high-level concept to a set of lower-level nodes.

\textbf{Supports} and \textbf{Refines} are used for different purposes. Support arcs intuitively explain "why" a concept exists: because another concept uses it. Refinement arcs explain "how" a value is found for a concept: by using the values of lower-level computations. In the example of Figure 3, which is drawn from the example of section 1.4, training-mode $\Rightarrow_s$ operator, but training-mode-value-set $\Rightarrow_r$ training-mode.

While the graph described by any horizontal layer of concepts may include loops (a concept may recursively support itself), the refinement in the vertical direction forms a strict hierarchy of trees (no concept may refine itself). This constraint is represented as

$$\forall n \in \textbf{Nodes}: \neg(n \Rightarrow^* R n)$$

where $\Rightarrow^*$ has the standard meaning of a transitive relationship: no node refines itself, either directly or transitively.

It is convenient to think of the refinement to lower levels of the conceptual map as the creation of a new problem space at the lower level. The new space gives boundaries to the part of the design described at this level. The formalism of creating
a new sub-space links directly to Stefik's work with operators to create new problem
spaces [42]. This provides a formal basis to describe operations to create new nodes
automatically, and new spaces to contain these nodes within a level of a conceptual
map.

Designer keeps levels of detail separate by identifying a concept at a low level
of detail as the refinement of a specific, higher-level concept. Distinctions between
subsets of nodes at the same level, and the identity of a subset that refines a concept,
are not currently used in the theory. Sublevels are therefore not specifically
identified.

This is partly because levels of abstraction are not cleanly partitioned in the
conceptual map. The notion of a "level of abstraction" is a fuzzy concept in a logical
sense: some nodes may not clearly belong to one well-defined level. The same
concept may in different contexts be treated as a lowest-level, primitive concept, or as
an abstract concept to be refined. If a logical level were actually defined as a
subgraph on the total space of concept nodes, it could include members from different
physical depths in the network.

The formalism does not give strict rules for choosing a refinement arc over a
support arc in any particular instance. Therefore, the constraints on concepts are
permitted to refer to nodes found along either type of relationship. Since the
refinement relationship is introduced for human convenience, rather than out of
computational necessity, an alternative interpretation of the relationship could be as a
view operator, creating a new sublevel to restrict the scope of visible supporting
nodes.
4.4.3 Values, Domains, and Types

Each concept models some entity or class of entities in the real world. Each concept is associated with a value. The value of a concept is interpreted by an intended mapping to its reference in the real world.

Since different concepts refer to different entities or classes, the value of each concept is drawn from a domain of values appropriate to the concept. This makes it natural to associate each concept with a data type. Although a type can be easily associated with each concept, no use is currently made of types, and a type system is not formally defined. Domains are defined only sufficiently to support operations on the values of concepts.

Let a set of domains define the domains of interest to which concepts can refer. Each domain is defined to be a set of values. Using the standard notation of set theory:

\[
\text{Domains} = \{\text{Domain}_1, \ldots \text{Domain}_n\} \\
\text{Domain}_i = \text{Values} \\
= \{v_1, \ldots v_n\}
\]

The interpretation of the values in each domain is determined by the intended mapping of the concepts to the real world. The domains of interest include such pragmatic values as input and output between the problem setting under consideration and the rest of the world. Some standard domains, such as Boolean = \{T, F\} and Natural = \{0, 1, \ldots\}, are assumed.
For every node in the set of nodes of a conceptual map, there is a value belonging to some domain in Domains. A DomainMap function can be taken to associate each node with its intended domain, and a ValueMap function to associate the node with a value in the domain appropriate to that concept.

\[
\text{DomainMap: } \text{Nodes} \rightarrow \text{Domains} \\
\text{ValueMap: } \text{Nodes} \rightarrow \text{Values} \\
\forall n \in \text{Nodes}: \text{DomainMap}(n) = \text{Domain}_i \in \text{Domains} \land \\
\text{Domain}_i = \text{Values} \land \\
\text{ValueMap}(n) = v_i \in \text{Values}.
\]

Operationally, at any given time in the design process, the value of a node may not be known. The value of a node is represented by a variable. If sufficient knowledge is not yet available to compute the value, its variable is unbound. When the value is successfully computed, its variable becomes bound to the actual value \(v_i\).

4.4.4 Annotating the Conceptual Map

The nodes and arcs of the conceptual map represent statements about the supporting relationships found in the structure of a design. The map is annotated with additional knowledge about the design. The most important class of annotations is constraints that define the computation of values for each concept. A few additional annotations are described in section 5.2.3 to attach bookkeeping attributes to concepts.

Dasgupta describes a design as a network of constraints. A conceptual map, on the other hand, describes a design as a network of concepts with attached constraints. The two descriptions are equivalent.
Constraints hold on values. The original concepts of the problem setting represent values from the "natural" domains in Domains. Even a value can be described as a constraint: it is easy to constrain a node to some constant value. Therefore, every concept of a conceptual map can be represented as a constraint.

Conversely, every constraint must be computed in order to be tested, and every constraint has a value (from the domain of Boolean truth values). Thus, every constraint can be represented as a concept. It seems more intuitive to use the concept as the fundamental building block.

For each node, there is an associated constraint. Each constraint is a relation on the value of the owner node and some set of other values:

\[ \text{Constraint}_i = <v_i, \ldots v_j> \]

The value of a node is usually related to the values of its supporters. For each \( v_x \), either \( v_x \) is a constant in some domain, or \( v_x = \text{ValueMap}(x) \) for some node \( n_x \). Then either \( n_x \Rightarrow_s n_i \) or \( n_x \Rightarrow_r n_i \). Section 5.7 addresses constraints on constant values and inputs from outside the problem setting.

In the example of section 1.4,

\[ \text{Constraint}_{\text{applicable-doctrine-statement}} = <v_{\text{applicable-doctrine-statement}}, v_{\text{operator}}, v_{\text{doctrine-option}}>. \]
Constraint$_i$ represents a relation defined specifically for node $n_i$. A function associates each node with its constraint from the set of all constraints:

\[
\text{Constraints} = \{\text{Constraint}_1, \ldots, \text{Constraint}_n\}
\]

\[
\text{ConstraintMap: } \text{Nodes} \rightarrow \text{Constraints}
\]

In this definition, the Constraint$_i$ for node $n_i$ includes a distinguished value $v_i$ that represents the value of $n_i$. The value of a node is equivalent to the value of its $v_i$ when its constraint evaluates as true. There is an evaluation function to evaluate a constraint and map it, when it evaluates to true, to the value of its concept node:

\[
\text{Evaluate: } \text{Constraints} \rightarrow \text{Values}
\]

\[
\text{Evaluate(Constraint}_i) = \text{ValueMap(node}_i) = v_i \in \text{Values}
\]

Constraints have been defined in terms of values. In the notation, symbolic variables stand for values. This carries through to the user syntax and implementation of Designer, where variable names are used to represent values in constraints.

To evaluate a constraint, some procedure must recognize if the constraint represents a relationship that actually holds on specific values. This is further complicated by the possibility of unbound variables as some of the arguments to the constraint.

Evaluating a constraint is intended to compute a value for a single variable representing the value of the associated concept node. This suggests that it may be
useful to represent a constraint as a function. This is the intuitive view used to communicate with a user in Chapter 5:

\[ v_j \times \ldots \times v_k \rightarrow v_i \]

It is possible that more than one of the values related by a constraint may be unbound variables, to be computed during the evaluation of the constraint. It is also possible that the evaluation may require the computation of intermediate values.

In the simplest case, a constraint describes a relation among constants. For example:

\[ \text{add}(2, 1, 3) \]

or, with a single unbound variable representing the value to be computed for the constraint's node,

\[ \text{add}(2, 1, v) \]

In this example, an evaluation procedure could look up the constraint in the set of literal instances that defines this relationship. In general, it is necessary to define each constraint in terms of simpler relationships that can be directly evaluated.

A constraint must therefore be defined to have structure in addition to the simple statement of a relationship on values presented thus far. That simple statement can be interpreted as a head goal for the constraint (equivalent to the signature or protocol of a conventional procedure). In addition, a constraint has a defining set of subgoals, which form the constraint body.
A constraint evaluates to true when each element of its body evaluate to true. A constraint may fail at any step in its evaluation. However, it is convenient to talk about choices as a condition and result. For this reason, the body of a constraint is split into a guard and a "proper" body. The body is often procedural in effect, as when it performs a calculation and creates a value. The guard insulates the more declarative conditional test from the more procedural body. The distinction does not affect the computational model. It provides practical benefits for code generation without considering backtracking, which are discussed in section 5.12.

The structure of a complete constraint is defined as follows:

\[
\text{Constraint} = <\text{head}, \text{guard}, \text{body}>, \text{where} \\
\text{guard} = \{\text{head}_i, \ldots \text{head}_j\}, \\
\text{body} = \{\text{head}_k, \ldots \text{head}_l\}.
\]

That is, the definition of a constraint is a head and a collection of additional heads, which function as subgoals. Each head describes a relation on values as described above:

\[
\text{head} = <\text{argument}_1, \ldots \text{argument}_m>, \text{where} \\
\text{argument}_i \text{ is either a variable or a constant value.}
\]
For example, the constraint in the example of section 1.4 is defined as a relation on values, but has a structure of subgoals that defines in what way those values are constrained:

\[
\text{Constraint}_{\text{applicable-doctrine-statement}} = \langle v_{\text{applicable-doctrine-statement}}, v_{\text{operator}}, v_{\text{doctrine-option}} \rangle. \\
= \langle \text{Constraint}_{\text{doctrine-template}}, \text{Constraint}_{\text{operator}}, \text{Constraint}_{\text{insert-attribute}} \rangle, \text{ where}
\]

\[
\text{Constraint}_{\text{doctrine-template}} = \langle v_{\text{doctrine-option}}, v_{\text{doctrine-template}} \rangle, \\
\text{Constraint}_{\text{operator}} = \langle v_{\text{operator}} \rangle, \\
\text{Constraint}_{\text{insert-attribute}} = \langle v_{\text{doctrine-template}}, v_{\text{operator}}, v_{\text{applicable-doctrine-statement}} \rangle.
\]

The heads form a disjoint set, in the sense that each represents a unique instance of a computation. Multiple occurrences of a head in the body of a constraint are interpreted intuitively as performing the same computation for different inputs. Literally identical multiple occurrences are legal, but usually of little practical use. A repetition of the constraint head in its body is interpreted as a recursive computation, which would be reflected structurally by a loop in the conceptual graph (a node is its own support). Intuitively, a literally identical repetition of the constraint head in its body would suggest an infinite loop.

The values bound to the variables in a constraint can be supplied from two sources. The constraints in the body of a constraint contain variables that reference either values of supporting nodes or values of primitive, built-in constraints. When there are references to more than one supporting nodes, some of these variables will play the role of intermediate results local to the constraint.
In addition, as discussed in section 4.4.6, it may be necessary to provide a constraint with provide values that are not included in the support nodes. A constraint is associated with a set of input parameters, which communicate data between a calling constraint and a called constraint. A constraint computes a value, which is also communicated to its calling constraint as the result of its execution. Data is communicated from one branch of a computation to another when the constraint of a node (that is the ancestor of both branches) computes a value for one branch, and supplies it as an input parameter to the other branch.

In particular, to gain Turing-expressiveness, global data is necessary; therefore, every calling constraint will always supply at least its global data. A constraint may add some of its local variables to either the head or the tail of the portion of the variable list interpreted as global data. In section 4.5, this list of variables is naturally interpreted as a concatenated set of strings, by analogy with the infinite tape of a Turing machine.

Each value in a constraint must be distinguishable in the relationship. The simplest way to accomplish this is to assume an ordering on the arguments. The values associated with a constraint are represented as an ordered list of variables, which can be read or written. A constraint can form any appropriate sublist to be used as input arguments for a constraint it evaluates. The arguments of a constraint refer to exactly those values on which the constraint defines a relationship.

A notational syntax will be needed in the implementation of the Designer tool for users to denote constraint heads. It is convenient to identify each constraint by the name of the concept to which it belongs. This will give a syntactic style like
those common in applicative and functional programming. If the head goal is visualized informally as:

\[ \text{identifier}(	ext{argument}, \ldots \text{argument}) \]

the arguments represent the other values with which the owner concept’s value is related by the constraint. If one argument represents the constraint value, and the global data is accounted for, all of the remaining arguments must be used as input arguments. The example of section 1.4 is restated in this syntax as:

\[
\text{applicable-doctrine-statement}(\text{applicable-doctrine-statement}, \text{operator}, \\
\text{doctrine-option}) = \\
\text{doctrine-template}(\text{doctrine-option}, \text{doctrine-template}), \\
\text{operator}(\text{operator}), \\
\text{insert-attribute}(\text{doctrine-template}, \text{operator}, \\
\text{applicable-doctrine-statement}).
\]

The values related by a constraint are described formally as follows. Input parameters are distinguished with the identifier function Input, and a node is associated with the values its constraint relates by the function ValueMap. It is unnecessary to distinguish here between bound values and unbound variables.
\( \forall n_1, \ldots, n_k \in \text{Nodes} \exists \text{Values}_i \in \text{Values}, \text{Constraint}_i \in \text{Constraints}:
\)

\[
\text{ConstraintMap}(n_i) = \text{Constraint}_i \land \\
\text{ValueMap}(n_i) = \text{Values}_i \land \\
(\forall v_x \in \text{Values}_i; \text{ValueMap}(n_x) = v_x \lor v_x \in \text{Input}(n_i)) \land \\
(\forall x \in \{j \ldots k\}; n_x = n_i \rightarrow \text{ValueMap}(n_x) \in \text{Values}_x) \land \\
(\text{Constraint}_i(v_i, v_j, \ldots v_k) = T \rightarrow \text{ValueMap}(n_i) = v_i).
\]

Some constraints hold on the variables. If the constraint for \( n_i \) evaluates the constraint for \( n_j \), then \( n_j \)'s value is among the values of \( n_i \). Also, \( n_j \)'s input arguments are a subset of the values of \( n_i \). The global variables are always among every node's input arguments. The scope of a variable is the constraint within which it is used.

Let there be a set of primitive, built-in constraints, that are assumed to be evaluable without further development. The primitives explicitly include one that is equivalent to the \text{ValueMap}() function to access the values of supporting nodes. This is used by a constraint to trivially retrieve values of its lower-level supporting nodes.

Because all simple combinations of values of supporting nodes will be expressed as primitives, and supporting values can be accessed and "passed through" any number of levels, it is sufficient to allow all subgoals in the body to be drawn from the set of primitive constraints.

This definition permits a constraint to give something other than a one-to-one mapping. This makes constraints represent a more general computation. It will also be useful later, when code is generated by mapping constraints onto Prolog clauses.
4.4.5 Semantics of the Conceptual Map and Goals

It is difficult to point to a single meaning for a complex problem setting. Intuitively, a problem setting has some significance to humans as a complete entity. The conceptual map serves as an abstract, or summary entity to express this significance. It captures a set of concepts and relationships too large and too complex for a human to manage mentally all at once.

However, a proposed or desired solution for the problem has an intuitive meaning. From Eden, a completed conceptual map contains a subset of concepts that represent the problem solution.

Any subset of nodes can be assigned a higher-level parent node that serves as an abstraction. Therefore, there is defined a unique concept that represents the highest-level problem solution. This unique concept is taken to represent the meaning of the entire conceptual map. The meaning of the map is defined to be the meaning (i.e., the value) of this unique concept.

\[
\text{Goals} \subseteq \text{Nodes}, \\
\text{Goal}_0 \in \text{Goals}, \\
\text{ValueMap(Graph)} = \text{ValueMap(} \text{Goal}_0 \text{).}
\]

The graph computes a solution value when it successfully evaluates the constraint of Goal_0.
4.4.6 Computational Meaning of Constraints

Since the arguments of a constraint are the values of nodes, each is represented by a variable, which may be either bound to an actual value or unbound. The value of a node is not known until its constraint has been evaluated to true. By analogy with the Prolog computational model [20], an unbound variable is defined to represent an unknown value.

There must be a procedure to effectively evaluate each constraint. Further, this procedure must handle the condition where some arguments (the values of supporting nodes) are not known.

Because each constraint is defined in a consistent form, a single procedure can evaluate all constraints. The structure of a constraint definition from section 4.4.4 defines each constraint in terms of either the values of its supporters or primitive, built-in constraints. The primitives are equivalent to axioms in a proof system or ground facts in a logic program. The evaluation procedure then uses the same goal-driven procedure as Prolog:

\[
\text{to evaluate Constraint}_i,
\]
\[
\text{evaluate each of its subgoals.}
\]
Since evaluating a constraint is defined as determining the value of its concept node, applying the ValueMap function to a node is equivalent to executing the evaluation procedure on the constraint. This is formalized as follows:

\[
\text{ValueMap}(n) = \text{Evaluate(ConstraintMap}(n)) \\
\text{Evaluate(Constraint}_i) = \\
\land \text{Evaluate(Constraint}_i): n \in \{i, \ldots, j\}, \\
\text{Constraint}_n \in (\text{guard}_n \cup \text{body}_n) \in \text{Constraint}_i
\]

A constraint may only evaluate the constraints of concepts that support its own concept:

\[
\text{Evaluate(Constraint}_j) \in (\text{guard}_n \cup \text{body}_n) \in \text{Constraint}_i \implies p_j \Rightarrow p_i.
\]

It is common to view a goal-driven computation as a search tree. To achieve complete computational expressiveness, it is necessary to communicate values between subtrees. In other words, not only does a parent node use the values of its children, it must be possible to make the value computed for one child available as an argument to the computation of another child. This is equivalent to saying that a node may use not only by the values of its supporting nodes, but by the values of its sibling nodes. The "input arguments" of a constraint, those that represent values supplied by the evaluation procedure, serve this function.

Values can thus span more than one branch of an and/or tree. Context is made visible by this mechanism across arbitrary portions of the computation. This is one of the features of the computational model that extends its power beyond and/or trees to that of a Turing machine.
Computationally, the problem solution Goal$_0$ also serves as a distinguished entry point to the graph. Its constraint is the first one evaluated. The computation is thus given a starting point, which also represents the highest-level goal of the computation. To satisfy the goal (compute a meaningful value), its supporting nodes must be satisfied as subgoals. This computation extends recursively to the leaf nodes of the graph. When all of the lower-level nodes have been satisfied (assigned values), the starting point can compute its own value, and thus be satisfied. The computation returns to its entry point to terminate.

In this computational model, constraints that evaluate to false leave their owner nodes with no value. As in Prolog, the evaluation procedure may attempt alternate definitions to evaluate a constraint. A concept whose constraint evaluates to false (meaning that none of its alternate definitions are true) has no value. In particular, if Goal$_0$'s constraint evaluates to false, no value can be computed for the map.

For a leaf node that depends on no other nodes, there must be a constraint that either obtains values from the world outside the graph, or encodes a knowledge base of some domain knowledge from which a value can be derived. This is a constraint whose only argument is the value of its owner node. Some constraints may involve the creation of output to an end user or calling system.

This entire computational model is derived directly from that of Prolog. For a formalization of the Prolog computational model, refer to Lloyd [20]. The advantage over Prolog is in the user's ability to describe structure and control in the graph itself before descending to the computational details.
4.4.7 Choice, Loops, and And/Or Structure

Eden's conceptual map is a general, potentially cyclic structure. For ease of computing, the map is modelled as an and/or graph. The notion of an and/or graph is derived from the more formal definition of an and/or tree (see Figure 11). An and/or tree is acyclic. To support iteration and recursion, the computational graph must permit cycles. The classic and/or tree is extended to a general directed graph (see Figure 12).

In a formal and/or tree, the child nodes of an and-node must be or-nodes, and the child nodes of an or-node must be and-nodes. Since human designers work with a problem in an informal way, it is desirable to postpone the rigorous enforcement of this pattern as long as possible.

![And/Or Tree Diagram]

**Figure 11:** And/Or Tree
Following common practice, this thesis also uses the term "leaf node" (which, strictly speaking, is accurate only in a tree) to describe a node in a graph that has no child nodes.

The combination of sequence and choice in a design implies the and/or structure of a conceptual graph. To a large degree, language syntax is not needed because the features of sequence and choice are expressed directly in the structure of the graph.
A simple constraint is defined in terms of the values of its supporting nodes. According to the computational model in section 4.4.6, the supporting nodes are evaluated sequentially to determine their values. The correct sequence for evaluation is determined by assuming that the subgoals in a constraint definition are ordered in their correct evaluation sequence. Such a node is an and-node.

Choice means that a node is constrained in more than one possible way. Each alternate definition of the constraint uses a subset of the available supporting nodes. Choices are made in the evaluation of a constraint by sequentially evaluating the alternate definitions, until one of them evaluates to true. The guard expressions in the constraints of a set of sibling supporting nodes provide a computation for their parent node's constraint to choose and branch among them.

Such a node is an or-node. Attempting alternate definitions if the first one does not evaluate to true requires additional support for backtracking, as found in Prolog. The evaluation procedure uses backtracking to search for solutions to a constraint until it finds one.

In a complete and/or tree, or-nodes can always be recognized by following the alternating pattern from the root and-node of the tree. For ease of human editing, it is not desirable to impose a strict and/or pattern throughout the tree. Alternatively, a Boolean attribute is annotated to or-nodes so they can be recognized.

Sibling guard expressions can, but need not, be mutually exclusive. Some computational models choose only one of the guards that evaluate to true, while others permit the choice of more than one. For convenience, this theory follows the lead of Prolog, and of Hoare's guarded expressions [15], to evaluate only the first procedure whose guard expression returns true. Similarly, this theory permits
nondeterministic computations. The encoding of choice primarily in the structure of the conceptual graph is intended to be more intuitive and abstract than the Prolog formalism of alternate clauses.

Cycles in the conceptual graph represent exactly those computations that are recursive. A loop is not directly dependant on the and/or structure. However, the header node (entry point) of a loop is clearly a choice point, and therefore an or-node. To avoid the complexity of defining loop identity and associating headers with specific loops, a node is constrained to act as a header for a maximum of one loop.

That constraint implies that loops are simple cycles in the conceptual graph - i.e., that loops contain disjoint sets of nodes. No expressive power is lost, since any graph structure involving a node in more than one cycle can be transformed to a set of simple cycles by creating duplicate copies of the needed subsets of nodes. The restriction is simpler, and therefore more understandable, than overlapping loops.

From basic recursive function theory [29], every iterative computation can be represented as a recursive computation. The constraints for the nodes in any loop in a conceptual graph are written recursively. The alternate constraint definitions for the header node of a loop correspond to the base case and recursive case of a recursive definition. The theory makes no provision for checking infinite loops.

Although a loop can always be detected in the structure of a graph, the header node of a loop is not so easily detected. It is therefore distinguished by annotating an attribute. If each node is constrained to be a header node for at most one loop, one bit of attribute information is sufficient to locate each loop and its header node.
4.4.8 Semantics of a Constraint

Constraints are annotations to the graph. There is no established theoretical body on the semantics of graph annotation. This work begins to explain their significance by definition.

Constraints are used to determine the value of a node, and to give more specific semantics to the interpretation of arcs on the conceptual graph. Hence, they enhance the expressiveness of the graph.

This is analogous to the practice of adding attributes to the context-free grammar of a computer language, thus increasing its computational power beyond context-free [1]. Also like programming languages, where the addition of grammatical attributes increases expressive power, the annotated nodes add expressive power to the basic and/or tree.

Eden lets the arcs between concepts be completely general notions of support. This is appropriate during problem formulation. The exact relationships between concepts are not yet clearly understood in the early stages. A graph with no semantics for its arcs can have no semantic constraints to check when a designer changes arcs experimentally.

Once a designer begins to add constraints to the graph, the arcs take on more specific meaning. An arc between two nodes means that those nodes are related by some constraint expression. Similarly the value of any node is constrained by some expression. The constraint on a node must define how the values of supporting nodes are combined to compute a value for the node.
Two models are combined in the conceptual graph. In the structural model of the map, it is natural to think of a value as an attribute of a concept. In the declarative logic model of constraint, the constraint is the most important entity, and its attributes are the values it relates. The two models are made to cooperate by identifying the value of a concept as a distinguished member of the set of values that are arguments to a constraint. This identification is sound because standard computability theory shows that the functional computational model is equivalent to the predicate calculus model (the basis for constraints) [29]. Any function returning a value can be expressed as a predicate, in which one argument represents the value of the function. Conversely, any predicate that might be intuitively interpreted to produce a result in more than one of its arguments can be rephrased so that a single argument represents a complex data structure containing all of the original results.

The meaning of constraints is intended to be declarative. Constraints permit designers to make statements about a design. Eventually, however, procedural details must be considered in the definitions of constraints. Success at remaining in a declarative abstraction depends on how completely procedural control constructs can be inferred from the graph structure and constraint heads. The definition of loops and choice in the graph structure (section 4.4.7) abstracts the most frequently-used procedural details away from the constraint definitions.

4.4.9 Semantics of a Concept

From database theory, the meaning of an entity is equivalent to the set of values of its attributes. Each annotation, or attribute, can be further annotated with an identity, or with the type of the attribute, according to which the value of the annotation should be interpreted. Since graph annotations are equivalent to attributes
on entities (concepts), the meaning of a concept can be taken to be the value of its constraint.

Equivalently, the knowledge in an annotation can be interpreted as a logical predicate on the annotated concept. The values of the annotations are interpreted as arguments to the predicate. As in the attribute interpretation, the annotation must convey the name of the predicate, which is interpreted in a model interpretation sense as providing the meaning.

In a simple example, the entity "Procedure" in Figure 8 is described by its attribute "time-used", and its relationships to the entities "Sysproc" and "Module". The database theory interpretation of "Procedure" says that its meaning is the complete set of tuples that instantiate the template

\[
<\text{time-used, Sysproc, Module}>.
\]

The logical interpretation of the meaning of an entity is its mapping to the thing it references under an intended interpretation. The meaning of any isolated entity (atom, predicate, or function) is considered in the context of other entities related to it. The language of logic might interpret this entity as follows:

\begin{align*}
\text{node(Procedure)} \\
\text{attribute(Procedure, time-used)} \\
\text{entrance-for(Procedure, Module)} \\
\text{part-of(Procedure, Sysproc)}.
\end{align*}

Intuitively, the "meaning" of an entity is usually expressed as something simpler than the collection of all its relationships. Interpreting this entity as a concept
node, Figure 9 chooses to present its attribute as a supporting relationship, and the other two relationships as ones in which Procedure supports other nodes. From section 4.4.3, Procedure computes a value for itself from the value of its attribute.

The Computational Theory of Design defines the meaning of each concept to be its value. This is more concise than the database definition, and more object-like than the logic definition.

In this definition, a concept is a more central entity than its relationships. A concept node is defined to own its annotations. A data structure that reflects this view of the entity Procedure is the following:

```plaintext
node(Procedure
  annotations(value(variable-for-computed-value)
    constraint(compute-from(supporters))
  )
  supporters(time-used)
)
```
4.4.10 Computations on the Map

This section describes control operations on the objects defined in the previous sections. It defines metaknowledge about the design activities above the level of individual concepts and constraints. It adds new operations to create and manipulate the design. The result of applying these operations to a conceptual map is a completed design, which can then be executed to produce a value.

Pragmatically, executing a design requires a virtual machine that takes a design as input, and realizes it on an actual computer. Rather than building a new interpreter for the design model described herein, this thesis refines the design into an existing programming language. The value of the design is computed by executing its implementation as a program.

The completed design is a problem solution (in a weak sense: when executed according to its computational method, the design is an effective procedure for computing a problem solution). The design model provides a way to describe the problem setting. Applying the meta-operations to the model captured in a conceptual map produces a problem solution. The problem solution exists at a level of abstraction that does not match real machines; therefore, it must be refined further to a level of detail that can be directly executed.

The meta-operations on the design model are not very amenable to rigorous formalization. Exploration of effective operations on a design is the experimental portion of this thesis. An informal description of the implemented operations is found in the discussion of the Designer computer program, in Chapter 5. The formal
description of the theory can, however, supply some constraints on the experimental system.

A graph is defined to be fully annotated when there is a procedure associated with every concept. There is a decision procedure that determines if a node has its procedure, and one that determines if every node in the graph has its procedure. These are defined as follows:

HasConstraint: **Nodes** \rightarrow Boolean
HasConstraints: **Graph** \rightarrow Boolean = \bigwedge(n \in **Nodes** \in **Graph**: HasConstraint(n))

It may be necessary to inspect a procedure and to "meta-execute" it, to determine if it is complete. These concepts and actions are defined only by implementation.

One standard paradigm for transformation by a formal system is to move an intermediate state (a conceptual map under construction) toward a solution by iteratively operating on its components. This is a natural model of the human model of design used in section 4.2.1, in which the designer alternates between editing and reflecting. Designer, which gives interactive support to human designers, accepts the result of human reflection as an input for its edits. However, the automated system can also use the time expended by the human reflective cycle to do its own reflection. By applying the completion detection operations, Designer can determine which nodes must be edited before the design can be complete. Sometimes, it can construct edits to be applied, to move the graph automatically toward a more completed state.
This is opportunistic reasoning, in cooperation with human reasoning. An automated agent reflects on the conceptual graph in parallel with the human agent. The graph is continually changing because of the human’s actions. Whenever the system finds an opportunity to refine the graph, it takes the opportunity.

The editing cycle can be described procedurally, for multiple agents \( a \in \text{Agents} \), the community of cooperating agents, as:

\[
\text{Construct-Graph}(\text{Agents}, \, \text{Graph}, \, \text{Meta}_{\text{Design}}, \, \text{Meta}_{\text{Edit}}) =
\]

\[
\text{repeat until } (\text{HasConstraints}(\text{Graph}))
\]

\[
(\text{Inconsistencies} \, := \, \text{Reflect}(a, \, \text{Graph}, \, \text{Meta}_{\text{Design}}), \]

\[
\text{Edits} \, := \, \text{Construct-Edit}(a, \, \text{Inconsistencies}, \, \text{Meta}_{\text{Edit}}),
\]

\[
\text{Graph} \, := \, \text{Apply-Edit}(a, \, \text{Edits}, \, \text{Graph})
\]

or declaratively as:

\[
\text{Constructed-Graph}(\text{Agents}, \, \text{Graph}, \, \text{Meta}_{\text{Design}}, \, \text{Meta}_{\text{Edit}}) \leftrightarrow
\]

\[
\forall
\]

\[
\exists \, a \in \text{Agents} : \text{Reflect}(a, \, \text{Graph}, \, \text{Meta}_{\text{Design}}, \, \text{Inconsistencies}) \land
\]

\[
\text{Construct-Edit}(a, \, \text{Inconsistencies}, \, \text{Meta}_{\text{Edit}}, \, \text{Edits}) \land
\]

\[
\text{Apply-Edit}(a, \, \text{Edits}, \, \text{Graph}, \, \text{Graph}') \land
\]

\[
\text{Constructed-Graph}(\text{Agents}, \, \text{Graph}', \, \text{Meta}_{\text{Design}}, \, \text{Meta}_{\text{Edit}}).
\]

where:

A Computational Theory of Design
sequencing operator.

; = coprocessing operator. "," and ";" are used in the standard way for communicating sequential processes. The editing constraints can be applied in parallel by multiple agents.

Reflect: \( \text{Agents} \times \text{Graph} \times \text{Meta}_{\text{Design}} \rightarrow \text{Inconsistencies} \)

This operation applies the goals and constraints in \( \text{Meta}_{\text{Design}} \) to the nodes and arcs in \( \text{Graph} \), and produces metaknowledge, Inconsistencies, about subparts of \( \text{Graph} \) that are "wrong" according to \( \text{Meta}_{\text{Design}} \).

Construct-Edit: \( \text{Agents} \times \text{Inconsistencies} \times \text{Meta}_{\text{Edit}} \rightarrow \text{Edits} \)

This operation maps the knowledge in Inconsistencies about undesirable states of \( \text{Graph} \) onto editing actions that can transform those states to more desirable states.

Apply-Edit: \( \text{Agents} \times \text{Edits} \times \text{Graph} \rightarrow \text{Graph} \)

This operation applies editing operations to \( \text{Graph} \), producing an updated version of \( \text{Graph} \).

\( \text{Agents} = \{a_1, \ldots, a_n\} \).

\( \text{Meta}_{\text{Design}} = \) metaknowledge about constraints on, and desired characteristics of, conceptual graphs.

\( \text{Meta}_{\text{Edit}} = \) metaknowledge about editing operations.
Inconsistencies = a representation of what an agent knows about Graph as a result of reflecting on Graph. Inconsistencies includes a list of subparts of Graph that do not satisfy the constraints and goals of MetaDesign.

Edits = a representation of actions to apply to Graph.

The value of a concept can always be interpreted as the logical truth value of some predicate about that concept. The equivalence of logical statements and computations provides an easy way to include operations on metaknowledge in ordinary computations of the system. Dasgupta uses this technique in his algebra of design constraints to provide operations on statements about constraints at one level of abstraction. These operations compute the plausibility of constraints found at a different level of abstraction. The current work has been limited to a tractable scope that does not consider more general types of metaknowledge. However, the needed knowledge structures exist to support possible future extensions.

This type of knowledge is more typically available in a knowledge base, either global or encapsulated within the part of the system that uses it, rather than being supplied as input parameters.

This simple description has achieved a formal representation of informal human processes, and shows how an automated (necessarily formal) system can cooperate with a human on the same task.

The formalization ignores questions of coordination. Obviously, the implementation must ensure that updates to the same graph by multiple agents do not result in an inconsistent graph. The identification of the agent performing each
operation has been made visible because this knowledge is necessary to solve the coordination problem.

Since many subparts of the design built by a human are likely to be informal, one effect of the Construct-Edit operation is to represent those parts more formally. This realizes the third part of Schön’s act/reflect cycle: the result of reflecting on actions is to bring more of the problem domain within the domain theory. This will enable actions on future iterations of the cycle to be executed more efficiently and reliably.

Here, the theory being extended is the very specific theory of the design under construction. Schön discusses the more general activity of extending the theory of a professional domain; therefore, the new knowledge can be applied by different professionals to different problems. Designer stores none of this knowledge after its use on one problem. Designer is therefore a weak (but interesting) implementation of Schön’s theory of design.

4.5 COMPUTATIONAL COMPLETENESS OF THE MODEL

This thesis claims that a computational model based on the and/or tree, with augmentations, can be used to design any software artifact. To know that this system is really useful for general software development, it is necessary to know that the augmented and/or graph is powerful enough to handle any general computation.

In computability theory, it is common to show the equivalence of two computational models by proving that each can perform any computation possible by the other. This thesis only shows that an augmented and/or graph is as powerful as a
standard measure of "powerful enough to do useful work." The "other direction" of the proof, that the standard measure is equivalent to an augmented and/or graph, is peripheral to the purposes of the thesis.

The standard measure for the power of a computational model is the Turing machine. This thesis takes for granted (as is usual) Church's hypothesis that the Turing machine is the most powerful computational model possible [29].

There are several ways of doing computations equal in power to the Turing machine. If some computation can be performed at all, it can be performed by the Turing machine or any Turing-equivalent computational model. If the and/or graph is as powerful as a Turing machine, it can be used with the confidence that it can handle any design task.

This section contains a formal proof that the augmented and/or graph used to implement the Computational Theory of Design is equivalent to a Turing machine. This proof does not say much about the practicality, efficiency, ease of use, or appropriateness of the design support tool built as part of this thesis. It does, however, give an important assurance that the tool is powerful enough to do real work.

It has been proved that a simple and/or tree is computationally equivalent to a context-free grammar [13]. This section develops a proof that adding cycles and attributes to the nodes of the tree, and sharing those attributes between branches, enhances its expressiveness to that of a phrase structure grammar. Analogy is drawn to Prolog itself. The attributed nodes of an and/or tree are equivalent to the variables that Prolog makes available, by backtracking, to alternate branches of its own proof tree.

A Computational Theory of Design
Theorem 1. The augmented and/or graph described is equivalent to a Turing machine.

Proof. The theorem is proved if it can be shown that any computation that can be performed by a Turing Machine is also performed by the augmented and/or graph. This is done by showing that a complete simulation of a Turing machine can be constructed with an and/or graph. A complete simulation is achieved when each structural part of a Turing machine is represented in an and/or graph, and each action possible by a Turing machine on all inputs are implemented in the actions possible by the and/or graph.

4.5.1 Formal Description of the Turing Machine

The standard formalism for a Turing machine is used. A complete explanation of a Turing machine would be more detail than is necessary. However, since it is necessary to demonstrate that each part of the Turing machine can be implemented in an and/or graph, it is necessary to describe each component of a Turing machine. The traditional formalization is as follows:

Let \( T = (K, V, Z, M, q_0, H) \) be a Turing machine. In the formalism,

\[
K = \text{a finite nonempty set of states.}
\]

\[
V = \text{the alphabet of input symbols. } V \subseteq Z. \text{ The additional symbols of } Z \text{ described below are explicitly not in } V.
\]
\[ Z = \text{the alphabet of symbols available to } T. \quad Z = \mathcal{V} \cup \{\text{state descriptors}\} \cup \{\square\}. \quad \square \text{ represents a blank on the tape.} \]

\[ M = \text{the state transition function for } T. \]

\[ q_0 = \text{the start state, a member of } K. \]

\[ H = \text{the set of final states. } M \text{ contains no transitions from states in } H. \]

Besides the components shown above, the Turing machine \( T \) is embedded in a computational context: its tape and state control procedure. The tape alphabet \( Z \) is only meaningful in this context.

The tape is an infinite memory for intermediate states. To execute a recursive computation, it is necessary to remember an indefinite number of entries to the computation. The tape alphabet \( Z \) contains a language representing the memory stored on tape. This is a different language than that used to represent the inputs, \( T \). It includes sufficient unique symbols (the state descriptors of \( Z \)) to represent all the possible states of \( K \).

If a Turing machine is interpreted as an acceptor for a Type 0 language \( L \), then the intermediate states of the computation represent the nonterminal symbols of the Type 0 grammar. \( V = \mathcal{V_T} \), the terminal symbols of \( L \): the strings accepted by \( T \) are the same as the strings formed in terminal symbols of \( L \). \( Z \) includes \( \mathcal{V_T} \) and \( \mathcal{V_N} \), the nonterminal symbols. The special symbol \( \square \) (blank) in \( Z \) represents the unwritten portion of an infinite tape.

A Computational Theory of Design

117
Additional, "housekeeping" symbols may be convenient to distinguish states stored on the state, and to recognize locations on the tape while navigating it to search for those stored states. A set of standard housekeeping programs is needed to scan back and forth on the tape to locate various saved states. The tape alphabet may be represented by a single or many symbols. None of these details are needed for the proof. Neither are the distinctions between a deterministic and nondeterministic Turing machine.

Many texts (for example, [29]) contain standard techniques for developing these details of housekeeping, and using them in the descriptions of Turing machines. Those sources also contain proofs that the power of Turing machines and the correctness of proofs about their properties are not affected when the details are ignored.

The state control procedure of a Turing machine (analogous to the state transition function of a less powerful automaton) represents algorithms for carrying out the computation to be achieved by the machine. Since any significant computation depends on multiple steps that are simpler than the complete computation, the transition function M represents a collection of procedures. Some of these may be recursive, and some are housekeeping procedures that search for specific locations on the tape.
These transitions are described in terms of not only states, but also configurations. A configuration of a Turing machine at any moment describes a current state in \( M \), as well as the contents of the tape. The representation \( UqW \) describes a Turing machine in state \( q \), with the tape to the left and right of the machine’s read/write head represented by \( U \) and \( W \). The read/write head reads the symbol immediately to its right; therefore, \( W \) is typically represented as \( yW \) for some symbol \( y \). The form of entries in \( M \) is:

\[
M(q, y) = (p, z, \{R, L\})
\]

This expression describes a transition from state \( q \) to state \( p \) when the machine reads symbol \( y \). It replaces \( y \) with symbol \( z \), and moves either right (R) or left (L) on the tape.

From standard theory [29], a small set of types of transitions is sufficient to perform all of the changes of state (equivalently, to perform all of the actions) that are required to perform all possible computations. These types of transitions are enumerated as follows:

a. \( UqyW \Rightarrow UzpW \) iff \( (p, z, R) \in M(q, y) \)
   On reading symbol \( y \) in state \( q \), replace \( y \) with \( z \) and move right.

b. \( Uq \Rightarrow Uzp \) iff \( (p, z, R) \in M(q, \uparrow) \)
   At the right edge of the tape, add a new symbol \( z \) and move right.

c. \( UxqyW \Rightarrow UpxzW \) iff \( (p, z, L) \in M(q, y) \)
   On reading symbol \( y \) in state \( q \), replace \( y \) with \( z \) and move left.
d. \( Uxq \Rightarrow Upxz \) iff \((p, z, L) \in M(q, L)\)
   At the right edge of the tape, add a new symbol \(z\) and move left.

e. \( qLW \Rightarrow zpW \) iff \((p, z, R) \in M(q, L)\)
   At the left edge of the tape, add a new symbol \(z\) and move right.

f. \( qyW \Rightarrow pLzW \) iff \((p, z, L) \in M(q, y)\)
   At the left edge of the tape, reading symbol \(y\) in state \(q\), replace \(y\) with \(z\) and move left across a blank.

g. \( qLW \Rightarrow pLzW \) iff \((p, z, L) \in M(q, L)\)
   At the left edge of the tape, add a new symbol \(z\) and move left.

### 4.5.2 Formal Description of the And/Or Graph

A simple formal representation for a general graph is as follows:

\[
\text{Graph} = (\text{Nodes, Arcs}), \text{ where}
\]

\[
\text{Nodes} = \text{a set of nodes}
\]

\[
\text{Arcs} = \text{a set of arcs between nodes. But the arcs form a relation on the nodes:}
\]

\[
\Rightarrow = \text{a mapping from Nodes into Nodes.}
\]
In an and/or graph (or tree), the interior nodes represent intermediate states of the computation represented by the graph. The nodes are completely analogous to the Turing machine states. Some nodes represent solutions to the computation (in a tree, the single root node). These nodes are analogous to the final states $H \subseteq K$ of the Turing machine.

The single start state, $q_0$, of a Turing machine maps initially to many possible start states of an and/or graph. The computation of the graph may begin with any leaf node. However, it is trivial to designate a new starting node in the graph, which chooses nondeterministically one of the original starting nodes. Since determinism has no effect on the power of the Turing machine, this issue has no effect on the proof. The real or created single starting node of the graph is thus equivalent to the defined start state of the Turing machine.

From standard computability theory [29], any computation may be interpreted as the acceptance/rejection of an input string by a grammar. Using this interpretation for both the Turing machine and the and/or graph, the input required by the computation is easily cast as a string written with the input alphabet $V$. Also in both cases, the same computational alphabet $Z$ contains $V$ plus sufficient extensions to describe the intermediate states, and to identify previously-unseen regions of the state memory.

The state memory is, for the Turing machine, its tape. In the and/or graph, the state memory is the set of variables that can be associated with each node. From the common practice of implementing an and/or tree with stack-based memory, it is easy to see that the state memory of the graph is at least equivalent to the stack of a pushdown automaton, or to the left-hand part of the tape of a Turing machine. It remains to be shown in the description of the state control of the and/or graph that its
variables can implement the full tape, or the dual stacks of a Turing machine or two-pushdown automaton.

Consider a graph computing some node/state \( q \). The computational model has been described as making a standard transit through the graph, visiting the dependents of each node in the order defined by the computation annotated to that node. The value of the node is computed and annotated only after all needed child nodes have been computed, and their values annotated. The possible states of the computation are represented as strings, formed by concatenating the values of the child nodes with identifying symbols. These strings may include unique values identifying if the child computations have been completed (it is irrelevant to this proof if that information must necessarily, or can conveniently, be encoded separately). The configuration thus far is a state \( q \) reading symbol \( y \), which is represented as:

\[ qy \]

Depending on the control information encoded in \( y \) about which child values have been computed, \( q \) will transit to a state to either

a. Compute a child node;

b. Use its children's values to compute its own value; or

c. Add its own value to the available state information for its parent, and resume its parent's computation.

By convention, consider the control algorithm to "move right" when it descends to a child node, and to "move left" when it returns to a parent node. For
every transition, the and/or graph reads a symbol representing the current state, writes a symbol representing at least the addition of the new move, and possibly a value annotation, and moves right or left. Therefore, the graph represents the tape of a Turing machine, its movements left and right, and its ability to add new states to the right end of its tape. It remains to be shown that the graph can add states to the left end of its "tape."

### 4.5.3 Equivalence of the Descriptions

Since the formal expression of a finite state automaton is a specialized graph, the formalization of a graph given in section 4.5.2 can be rephrased in a form closer to that of an automaton, as follows:

Let \( \text{Graph} = (K, V, Z, M, q_0, H) \) be an augmented and/or graph, where

\[
\begin{align*}
K & = \text{a finite nonempty set of states.} \\
V & = \text{the alphabet of input symbols. } V \subseteq Z. \\
Z & = \text{the alphabet of symbols available to } \text{Graph}. \quad Z = V \cup \{\text{state descriptors}\} \cup \{\bot\}. \quad \bot \text{ represents an undefined variable, corresponding to a blank on the Turning Machine tape that the variables simulate.} \\
M & = \text{the state transition function for } \text{Graph}. \\
q_0 & = \text{the start state, a member of } K.
\end{align*}
\]
\[ H = \text{the set of final states.} \]  
\[ M \text{ contains no transitions from states in } H. \]

It was shown in section 4.5.2 that \( K, V, Z, q_0, \) and \( H \) implement exactly their analogs in the Turing machine. \( M \) contains entries of the appropriate form, as follows:

\[ M(q, y) = (p, z, \{R, L\}), \text{ where} \]

\[ q, p = \text{nodes in the graph.} \]

\[ y, z = \text{concatenated string of variables annotated to a node.} \]

\[ R = \text{move right in the string of variables, so that new annotations for the new state are written to the right of those for the old state.} \]

\[ L = \text{move left in the string of variables, so that new annotations for the new state are written to the left of those for the old state.} \]

A string of state variables for the and/or graph is thus equivalent to the tape of a Turing machine, and the graph can scan left and right on the string of variables. For each of the three computational states listed above, the transition function will read the existing state, transition to a new node, and write a new state on the string of variables.

When transiting from a node, the choice of new node is always determined by the state transition function. The choice of moving left or right in memory (the string of variables) must be determined by the transition function independently of the
choice of the new node (new state). Yet conventionally, the system described moves right on descent to child nodes, and left on return to parent nodes.

The independence of these two choices in the Turing machine can be interpreted intuitively as reading and writing to any random location in global memory, at any time. The choice of moving left or right in memory would preferably be tied to the choice of next node to impose variable scoping and block structure on the computation.

This is done for practical reasons: empirical practice in software engineering shows that computations are developed more reliably under these restrictions. Unfortunately, the introduction of scope returns memory to a single-stack model. The full computational model for the and/or graph must allow an "escape" from scoped to global memory to gain the full power of a Turing machine.

The escape takes the form of global, or environmental, context variables. To gain full Turing-expressiveness, a string of variables of arbitrary and expandable size (a piece of the Turing machine tape) must be created when the start state $q_0$ is entered, and be made visible to every node entered. Since any state in the transition function may represent a utility procedure to search any location on the tape, a move right or left can be interpreted as a move into global memory, or out of global memory to return to scoped memory, without loss of generality. These is illustrated, using the syntax of Designer procedures, in Table 1.

The augmented and/or graph is now shown to implement the following seven types of transitions between configurations required by a Turing machine:
Table 1: Growth of Concatenated String of Variables

<table>
<thead>
<tr>
<th>Growth of concatenated state variables</th>
<th>A trace of procedures calling procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calling protocol:</td>
<td></td>
</tr>
<tr>
<td>procedure(input global variables, other inputs, output global variables, value)</td>
<td>local variables</td>
</tr>
<tr>
<td>procedure-1((global-1), (local-1-a), (local-2-c, global-1), value-1) (local-1-b)</td>
<td></td>
</tr>
<tr>
<td>/* procedure-1 passes the global data it receives on to procedure-2, expects and uses a single input data element, returns as a new global data list whatever is returned by procedure-2 */</td>
<td></td>
</tr>
<tr>
<td>procedure-2((global-1), (local-1-a, local-1-b) (local-2-a, global-1), value-2)</td>
<td></td>
</tr>
<tr>
<td>value-1 := some function of(value-2)</td>
<td></td>
</tr>
<tr>
<td>procedure-2((global-1), (local-2-a, local-2-b), (local-2-c, global-1), value-2) (local-2-a, local-2-b, local-2-c)</td>
<td></td>
</tr>
<tr>
<td>/* procedure-2 uses the global data it receives, expects and uses two input data elements, prepends a new value to the global data list */</td>
<td></td>
</tr>
<tr>
<td>value-2 := some function on(global-1, local-2-a, local-2-b, local-2-c)</td>
<td></td>
</tr>
</tbody>
</table>

a. \( UqyW \Rightarrow UzpW \) iff \((p, z, R) \in M(q, y)\)

On reading symbol \(y\) in state \(q\), replace \(y\) with \(z\) and move right. If the new node \(p\) is a child node, \(W\) may be interpreted as memory that
was originally written while visiting other nodes earlier. It may also be interpreted as global memory.

b. \[ Uq\uplus \Rightarrow Uzp\uplus \text{ iff } (p, z, R) \in M(q, \uplus) \]
At the end of the string of variables, concatenate new information and move right. This is interpreted as the usual actions when visiting a child node. A special case is the first node visited. New information is added to the top of the stack of scoped memory. It may also be interpreted as global memory.

c. \[ UxqyW \Rightarrow UpxzW \text{ iff } (p, z, L) \in M(q, y) \]
On reading symbol \( y \) in state \( q \), replace \( y \) with \( z \) and move left. If the new node \( p \) is a parent node, \( Ux \) may be interpreted as memory that was written when last at that node, before transiting to node \( q \). It may also be interpreted as global memory.

d. \[ Uxq\uplus \Rightarrow Upxz\uplus \text{ iff } (p, z, L) \in M(q, \uplus) \]
At the end of the string of variables, concatenate new information and move left. If the new node \( p \) is a parent node, \( Ux \) may be interpreted as memory that was written when last at that node, and \( z \) as new annotation written for node \( q \) before leaving it. It may also be interpreted as global memory.

e. \[ q\uplus W \Rightarrow zpW \text{ iff } (p, z, R) \in M(q, \uplus) \]
At the beginning of the string of variables, add a new symbol \( z \) and move right. If the new node \( p \) is a child node, \( W \) may be interpreted as memory that was originally written while visiting other nodes.
earlier. The beginning of the string would then indicate that $q$ is $q_0$. It may also be interpreted as global memory.

\begin{align*}
f. & \quad \gamma \gamma W \quad \Rightarrow p \downarrow z W \quad \text{iff} \quad (p, z, L) \in M(q, y) \\
& \text{At the beginning of the string of variables, reading a symbol } y \text{ in state } q, \text{ replace } y \text{ with } z \text{ and move left, making a new empty space for additional information at the beginning of the string. In ordinary scoped memory, "moving off the top of the stack" can only be interpreted as adding new information to the first node visited, where no information was recorded at the first visit. Since that is an illegal event, this transition represents exactly the action that a Turing machine can perform which is beyond the ability of a pushdown automaton: extending its memory to the left to write new global information.}
\end{align*}

\begin{align*}
g. & \quad \gamma \gamma W \quad \Rightarrow p \downarrow z W \quad \text{iff} \quad (p, z, L) \in M(q, \gamma) \\
& \text{At the beginning of the string of variables, concatenate a new first symbol } z \text{ and move left, making a new empty space for additional information at the beginning of the string. This case is similar to case } f.
\end{align*}

This section has demonstrated by simulated construction that an augmented and/or graph can implement the components, tape store, state control, and essential transitions between configurations of a Turing machine. The graph is therefore equivalent to the Turing machine. □
CHAPTER 5
THE DESIGNER SOFTWARE TOOL

Designer is a program that automates the manipulation of problems at the design level. It follows on the work done in SITE, a system that automatically acquires classification heuristics and generates expert systems.

SITE supplies automated help for one of the generic tasks identified in work on general problem representation, heuristic classification. Designer uses the work done in SITE to gather basic domain knowledge from a human user. It goes beyond the heuristic associations gathered by SITE to solicit descriptions of computations. This is an evolutionary step. Heuristic classification is a form of computation. Designer permits a broader class of computation than SITE.

Designer users are designers. Users faced with a problem have some computational result in mind. They wish to describe their problem and a desired solution, and to transform the problem description to a computation that produces the solution.

Designer attempts to fit into human design activities. The Computational Theory of Design describes (in Chapter 4) how human designers alternate between thinking about a problem space and performing calculations to discover low-level design commitments. Designer alternates between supporting human exploration of a problem setting, and opportunistically seeking parts of the design that can be automatically extended to connect the upper and lower levels.
Logically, this can be thought of as two concurrent processes. One supports a designer’s browsing and editing of the conceptual map. A second searches the conceptual map, evaluates its completeness, and automatically extends it.

Physically, Designer schedules the two activities as alternating goals in a single process. Problems with interprocess synchronization and mutually exclusive access to the conceptual map were not easily solved with the available Prologs.

Designer is a designer’s assistant. The program and a designer cooperate to refine and extend the conceptual graph. The tool performs two types of work to support this theory. One is bookkeeping: supporting a human designer who explores, browses, and edits the design. The second is a cooperative effort to extend the conceptual map built by the human.

Although Simon describes problem spaces as created top-down, in a hierarchical fashion, most writers observe humans working from the bottom up as well as from the top down. Designer supports design from both directions. Design from the top down creates a structure for the problem, and a framework for the design. Alternatively, design from the bottom up creates detailed design decisions and commitments.

Human designers often have difficulty connecting upper to lower levels, and mapping bottom-level nodes onto top-level nodes. This is an activity to which the automatic, intelligent operations of Designer can be profitably applied. The Computational Theory of Design describes design as structure that contains details. Much domain knowledge concerning how to extend structure and how to fill in detail is available.
Designer is a (mostly literal) translation of the Computational Theory of Design to the concrete data structures and computational engine of Prolog. A consistent formalism supports all levels of detail on the same underlying data structure. Designer assumes that the process of software development is a continuum. The conceptual map of Eden is not different in type, only in detail, from the and/or computation tree of Prolog.

The lowest level of detail refines the design computation to source code that can execute on a computer. Given the completed problem description and computational description, Designer can then generate a Prolog program to perform the computation.

5.1 DESIGN OF DESIGNER

The structural architecture of the Designer program is shown in Figure 13.

Designer's entry point is its editors. They provide the primary view onto the conceptual map and its components. The major program modules are:

a. Editors for the major structural components: the conceptual map, a concept, a constraint on a concept, and the parts of a constraint.
Figure 13: Structure of Designer
b. The evaluator, which emulates the human activity of reflecting on the design.

c. The edit generator, which automatically creates editing actions that can be applied by the editors to correct problems discovered by the evaluator.

d. The code generator, which generates Prolog source code that implements the conceptual map.

The program Designer uses the conceptual map directly as a data structure. Its predecessor program, SITE, is limited to conceptual maps that can be transformed to trees. In Designer, the map is represented and manipulated as a more general directed graph. See Figure 1 for an example of a conceptual map.

Designer is implemented as 13,000 lines of Prolog source code. Approximately 3,000 lines are reused from SITE. It is written in the VPI Prolog syntax, using an object-oriented style. The editors make extensive use of method inheritance and polymorphism, which are easily implemented in Prolog via metapredicates. It presently runs in a prototype form, using a text-based graph editor. The features of the program are discussed in the following sections. Additional operational details are found in the User's Guide, in Appendix D.

5.2 KNOWLEDGE REPRESENTATION

Designer acquires and manipulates knowledge from designers. As with most AI programs, knowledge representation [45] is a central problem for the program.
This thesis uses similar representations, and similar levels of abstraction, for all these types of knowledge. This contributes to a smaller knowledge domain and control space. The most important types of knowledge in Designer are as follows:

5.2.1 Blackboard

Designer inherits SITE's blackboard. It is a keyed collection that provides access and storage operations. The blackboard is implemented as assertions to the Prolog factbase. It is used because Designer reuses SITE modules that use it.

5.2.2 Prolog Factbase

Permanent knowledge about a design that does not need to use the blackboard is asserted to the Prolog factbase. In any Prolog program, there is a tradeoff between carrying knowledge in data structures as variable arguments to predicates, or asserting it to the factbase. Data structures as variables are preferred when the entire structure is frequently used. Designer generally uses the factbase because it more frequently searches for a specific fact.
5.2.3 Conceptual Graph

A conceptual map is represented as a general, possibly cyclic, graph of nodes and arcs. The graph itself is represented as a Prolog data structure, carrying general information:

concept-map(name, nodelist, looplist, filename).

The actual collection of nodes is collected as needed from the Prolog factbase. The "nodelist" slot is unbound most of the time, but used in several places to carry a list of the names of all concept nodes.

The map "name" is for the user's convenience. The "filename" is the name of a file to which the entire conceptual map can be saved as Prolog source code. The "looplist" is a list of identifiers for the loops found in the graph. It is updated whenever it is accessed, but the structure of the map has changed.

The concept-map structure is used by all modules in the program. Therefore, it is always communicated as an argument to Prolog predicates, and never stored elsewhere.

5.2.4 Concept

Concepts of a design form the content within the design structure. Concepts are represented as nodes in a conceptual graph. Each node is implemented as a frame. Slots of the frame represent various attributes and annotations to the concepts:
concept(name, description, node-type, procedure, data-type, supporters,
supporteds, refiners, refineds, value).

The concept "name" and "description" are for the user's convenience. The "node-type" is one of {and-node, or-node, loop-header}. These three characteristics are implemented in one attribute, rather than in the way formally described in section 4.4.7 for coding convenience. The "procedure" is the concept's constraint. The "data-type" is not currently used, but provided for possible future extensions.

The remaining slots of a "concept" structure are derived attributes. They are not actually stored with the data structure, but computed only when accessed. The slots "supporters", "supporteds", "refiners", and "refineds" contain lists of the names of nodes linked to a concept by support and refine arcs, in the child and parent directions, respectively. The "value" of a concept is derived from the result of evaluating its constraint.

Dasgupta suggests the need for additional control and metaknowledge in addition to the concept nodes. Its purpose is to give plausible support for the correctness of design decisions. Dasgupta's Theory of Plausible Design is, at least partly, a theory of how a society of designers convinces itself that a design is correct.

This has not been addressed in the initial version of Designer. However, the data structures needed to support additional concept types, such as design history and justification for decisions, are in place. Even if not used to compute plausibility, this type of knowledge can provide insights to humans attempting to understand a design.
Frames are stored on a blackboard reused from SITE. The blackboard is updated whenever a frame is changed. In addition, the name of each concept is stored in the Prolog factbase in a predicate:

\[
\text{model(concept-name)}.\]

This predicate is required by code reused from SITE.

### 5.2.5 Relationships among concepts

The relationships extend in three dimensions: Along a "horizontal" plane is the two-dimensional web of concepts that describe a problem at some level of abstraction. Horizontal relationships are concept support. The hierarchical levels of abstraction are found along a "vertical" plane. A concept at a higher, more abstract plane is decomposed and refined into an equivalent collection of concepts at a lower plane. Vertical relationships are concept refinement.

These relationships are represented as links among the nodes in the conceptual graph. They form the structure of the design. They are implemented as predicates on concept frames:

\[
\text{supports(concept, concept)}.\]
\[
\text{refines(concept, concept)}.\]

This general representation permits future extensions to relationships such as classification/generalization, data structures, or design structure elements. The
problem of maintaining slots that reference each other is solved by representing relationships between frames in a separate rulebase of predicates on frames.

The predicates are stored as ordinary Prolog facts. The Prolog factbase is updated whenever a relationship changes.

5.2.6 Constraints

The algorithmic description of how to compute the value of a concept is implemented, Prolog-style, as the defining subgoals of a constraint:

```
concept-procedure(constant, name, guard, body, type, description, own-variables, assigned-variables, conceptName, input-variables, ownerConcept, return, variables)
```

The term "concept-procedure" is an attempt to make constraints more accessible to users most familiar with imperative programming languages. The "constant" slot is used for convenience. When a leaf node is constrained to a constant value, that value is computed once and stored here, rather than carrying the constant constraint to be evaluated each time the value is needed. The "name" is an identifier, used when reflecting on the properties of constraints independently of their owner concepts. The "guard" and "body" are the two parts into which the defining subgoals of a constraint are split, as explained in section 4.4.4. The "description" is for the user's convenience. The "type" is not currently used. The "own-variables" is the list of variable identifiers for the arguments used by the constraint's own head goal.
The remaining slots are derived attributes, computed only when accessed. The slots "ownerConcept" and "conceptName" link a constraint to the name and the frame of its owner concept. The "variables", "assigned-variables", and "input-variables" are found by inspecting the constraint guard and body. They are lists of the identifiers of all variables used, of those that are bound to values, and of those that act as input arguments with values supplied by a constraint of a parent concept when it evaluates a supporting constraint. The "return" slot represents one of the values related by the constraint, which is identified as the value of the owner concept. It is derived from the value of the constraint body.

The guard and body of a constraint are each represented as a subgoal block:

procedure-block(body, type, assigned-variables, value, variables).

The "body" is a list of defining subgoals. The "type" is set to "Boolean" for guard blocks, but otherwise not currently used. The remaining slots are derived attributes, and are computed only when accessed.

The "variables" and "assigned-variables" are found by inspecting the constraint guard and body. They are lists of the identifiers of all variables used, and of those that are bound to values. The "value" slot represents the value of the constraint, which is the value of a distinguished argument among those related by the constraint. It is derived from the value of the block's last subgoal.

Each subgoal also has structure:

procedure-goal(name, variables, globalin, globalout, return).
This structure is used (because it is simpler) instead of a syntax tree when parsing the text representation of a goal. The "name" is the identifier, or functor (in Prolog terminology) of the goal. The "variables" is an ordered list of arguments to the goal. The two identifiers "globalin" and "globalout" refer to the global variables that are available to every constraint. "Globalout" is bound to "globalin" after any updates are made to it by the subgoal. It is only changed if the subgoal is one of the supplied primitives for that purpose. The "return" slot represents the last argument to the goal, which is identified as the value (what it returns, in functional terminology) of the goal.

Constraints are frames in their own right. Each is stored as a slot of its owner concept frame. Guards, bodies, and goals are each represented as a frame, but only stored within the larger context of a constraint frame.

Constraints currently are used only by concepts. The necessary framework is in place to associate constraints with other entities, such as if-used and when-needed slots, or simulation classes. None of these are used in the current version of Designer.

5.2.7 Domain Knowledge about Design

Designer reflects on a conceptual map by evaluating a knowledge base of constraints about the basic design features (the map, concepts, relationships,
constraints, and loops). This knowledge is represented as a declarative pattern of conditions and constraint predicates:

    design-constraint(name, applies-to, guard, body, priority).

The "name" slot is for convenience when comparing lists of constraints evaluated and not yet evaluated. The "priority" slot permits a simple, updatable control strategy. The only strategy currently used is to reflect on the most volatile features of a map first.

As with the guard and body of a concept constraint, each design constraint is split into parts for convenience. These parts are represented in the slots "applies-to", "guard", and "body". Most constraints apply to only one type of feature in a conceptual graph. The "applies-to" part of the constraint can be checked immediately by Prolog pattern matching. The "guard" is a conditional expression: the body of the constraint only applies if a condition is true. It is separated from the body for declarative clarity, as with the concept constraints. By making these slots references to the actual Prolog predicates that implement the parts of the constraint, it is also possible to reuse guard that can be applied to more than one constraint without copying the entire guard into each constraint data structure.

The design metaknowledge is stored as Prolog predicates, currently defined in the Designer source code. It could be extended for dynamic user editing in a future version, as discussed in section 8.5.
5.2.8 Domain Knowledge about Prolog

Designer generates Prolog code to express a design. This requires knowledge about how to generate Prolog syntax to represent each of its supported design structures, data types, and procedural operations. This knowledge is represented as a logical knowledge base in Prolog. Its style is procedural. Knowledge about how to transform each map feature to equivalent Prolog code is explicitly coded, as in the following simplified example:

```prolog
generate-code(map, Graph) :-
    access(Graph, nodes, NodeList),
    list_iterate(NodeList, generate-code(node, Node)).

generate-code(node, Node) :-
    access(Node, name, Name),
    access(Node, own-variables, Variables),
    access(Node, constraint, Constraint),
    generate-code(header, [Name, Variables]),
    generate-code(body, Constraint).

generate-code(header, [Name, Variables]) :-
    text-arglist(Variables, VariableText),
    TextList = [Name, ', VariableText, ') :-'],
    listtostr(TextList, HeaderText),
    write(stdout, HeaderText).
```

The Designer Software Tool
5.2.9 Edit Actions

When the evaluator discovers design constraint violations, it constructs a list of notifications and sends it in a message to the edit generator. Each notification is a simple data structure:

[constraint-name, feature]

with the identifier of the relevant constraint and the data structure in violation. The edit generator matches this to a knowledge base of corrective actions. It builds a list of suggested edits, each a simple list in the form:

[command-name, feature, argument-list].

This is the same semantic form used by the HCI when it interprets commands from an interactive user. An appropriate editor that can apply the command is determined from the type of feature. The argument-list may include other features, determined by the edit generator from the context of the feature being considered.

5.2.10 Loop

Because loops are a feature of the conceptual graph at a higher level than individual concepts, they are represented as data structures in their own right:

concept-loop(entry-point, base-node, recursive-node, nodes).
The "entry-point" is the name of the concept node distinguished as the loop header node. Its node-type is constrained to identify it as such. The formal description of section 4.4.7 described alternate constraint definitions associated with this entry point. Designer uses the tools already built for refinement nodes, to put the base and recursive case constraints in separate, refining nodes of the entry-point. The "base-node" and "recursive-node" are the names of these nodes. The slot "nodes" is a list of names of nodes linked together in a cycle on the conceptual graph that make up the loop.

The frame representing each loop is stored on the Designer blackboard for consistent lookup by the evaluator.

5.3 OPERATIONS ON THE CONCEPTUAL MAP

The actions required in the Computational Theory of Design were described formally in section 4.4.10. The actions are part of a simple cycle of editing operations. That description is repeated here:

\[
\text{Constructed-Graph} (\text{Agents}, \text{Graph}, \text{Meta}_{\text{Design}}, \text{Meta}_{\text{Edit}}) \leftrightarrow \\
\text{HasConstraints} (\text{Graph}). \\
\lor \\
\exists a \in \text{Agents} : \text{Reflect} (a, \text{Graph}, \text{Meta}_{\text{Design}}, \text{Inconsistencies}) \land \\
\text{Construct-Edit} (a, \text{Inconsistencies}, \text{Meta}_{\text{Edit}}, \text{Edits}) \land \\
\text{Apply-Edit} (a, \text{Edits}, \text{Graph}, \text{Graph}') \land \\
\text{Constructed-Graph} (\text{Agents}, \text{Graph}', \text{Meta}_{\text{Design}}, \text{Meta}_{\text{Edit}}).
\]
The basic design paradigm used in Designer is to translate the formal model into a concrete implementation. The formal description of the editing cycle becomes the following procedural flow of control:

Construct-Graph(graph) =

/* Using an interactive user as one agent and the modules evaluate() and construct-edit() as a second agent,
*/

repeat:

/* Only the reflective actions of Designer are modelled here. The user’s reflections are invisible to Designer, and implicit in editing actions obtained from the user. */

inconsistencies := evaluate(graph)
edit-actions := construct-edits(inconsistencies)
/* Acquire edit actions from interactive user input. */

edit-actions := edit-actions + acquire-edits(user)

graph := edit(graph, edit-actions)

until has-constraints(graph) == true.

The formal operations are mapped onto Designer’s procedural actions in Table 2. In practice, each is implemented as a Prolog predicate. Prolog’s high-level logical nature permits the mapping to be direct and intuitive.
Table 2: Mapping of Formal Operations to Designer Actions

<table>
<thead>
<tr>
<th>Formal Operation or Object</th>
<th>Equivalent Designer Procedure or Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructed-Graph(Agents, Graph, MetaDesign, MetaEdit)</td>
<td>construct-graph(graph)</td>
</tr>
<tr>
<td>a \in Agents</td>
<td>Uses an interactive user as one agent and the modules evaluate() and construct-edit() as a second agent</td>
</tr>
<tr>
<td>Reflect(a, Graph, MetaDesign, Inconsistencies)</td>
<td>evaluate(graph) for Designer. User's reflective actions not represented.</td>
</tr>
<tr>
<td>Construct-Edit(a, Inconsistencies, MetaEdit, Edits)</td>
<td>construct-edits(inconsistencies) for Designer. User supplies edit actions via interactive user input.</td>
</tr>
<tr>
<td>Apply-Edit(a, Edits, Graph, Graph')</td>
<td>edit(graph, edit-actions)</td>
</tr>
<tr>
<td>HasConstraints(Graph)</td>
<td>has-constraints(graph)</td>
</tr>
</tbody>
</table>

The program's structure reflects the organization of objects that make up a conceptual map. The map and the constraints on concepts are used conceptually to record complementary types of knowledge about a design. Similarly, the map, its concepts, and constraints are viewed and edited using different editors.

As a result of reflection on the conceptual map by Designer and a designer, both can generate plans for edit actions. Designer applies edits from both sources to the conceptual map.

Any problem setting can be represented in Designer. Although it is targeted for software engineers, it is possible to use it for other domains in which the designed
artifact is not a computer system. The current prototype is much more usable with small designs.

The program is entered by executing the Designer source code under the VPI Prolog system. On startup, the program initially enters its conceptual map editor with an empty template for a conceptual map. By default, Designer constructs a map interactively from user inputs. It can also read a conceptual map saved from a previous design work session. The user provides command selections from Designer's menus, and fills in details of concept annotations (the text representation of constraints and the few other attributes) via simple text editors.

The products of Designer are a conceptual map, saved as a file for future editing, and Prolog source code to implement the conceptual map in executable form. A saved map is also Prolog source code, but contains data structures suitable for the program's internal representations.

Although Designer is a direct implementation of the Computational Theory of Design, it encapsulates most theoretical details in familiar analogies, so that no special user background is required. Conceptual maps are easily recognized as a structural device. They share this in common with other design modelling tools, such as structure charts. The features of the conceptual map are filled in by interactive editors. Templates of appropriate responses are displayed. Constraints are edited in a syntax familiar to users of traditional computer languages.

Both the computations associated with concepts, and the values they compute, can be used as data by the design process. Computations on the augmented graph attributes can identify, and even apply, further changes that the design requires. The refinement of a design is thus partially automated. Designer reflects on and extends
the conceptual map automatically. It permits user override of its proposed edits, but
does not require user intervention during its evaluation of features in the design.

Designer reuses SITE’s knowledge acquisition modules to discover how users
wish to change the structure of the map and add detail to concept nodes. Where
possible, design metaknowledge is used to present the user with a small number of
choices. Designer then generates the data structures needed by its internal
representation.

The following paragraphs give an overview of the editing operations supported
by Designer on a design. See the Designer User’s Guide in Appendix D for screen
layouts, command syntax, and additional details.

5.4 EDITING A CONCEPTUAL MAP

The conceptual map editor supports creation and deletion of the highest-level
features, concepts and loops. Operations on the entire map, such as saving and
loading the map as a file, are implemented here. The editor provides a view of all of
the concepts in the map. Newly-created concepts have no annotations, links to other
concepts, or other attributes. Support and refinement relationships are created in the
concept editor, working from either of the concepts they link.

The loop editor requests the user to supply the names of nodes to serve the
functional roles of entry point, base node, and recursive node. These are added to the
appropriate slots, and a frame representing the loop is added to the Designer
blackboard.
As the other agent-like modules in Designer, the evaluator, edit generator, and code generator, could not be run in parallel, they are triggered from this editor. The user's judgement is relied upon to request evaluation of the design at appropriate times.

From the conceptual map editor, a user can enter the concept editor to edit any of the concepts in the map.

5.5 EDITING A CONCEPT

The concept editor supports creation and deletion of the relationships and annotations associated with a concept. The user can also create or delete concepts here.

Relationships with other concepts can be created, deleted, or altered. Creating a new link automatically enters a relationship editor that prompts the user to supply the type and direction of the relationship. It supplies a pick list of other concepts that are available to participate in the relationship.

Choice is represented in the structure of the conceptual map. By default, a concept is interpreted as an and-node. Its constraint will then use references to its refining concepts as sequential subgoals. If the user sets the value of node-type in the concept editor to be or-node, then Designer will automatically refine the concept to a sublevel of refinement nodes, each of which has a constraint that serves as an alternate clause for the constraint of the parent node.
From the concept editor, the user can enter the procedure editor to edit the constraint annotated to the concept. Designer refers to the constraint annotations as procedures in order to present a more familiar syntax to users.

5.6 ANNOTATING A CONCEPT

The procedure editor supports the creation of the components of a constraint on a concept. The annotation is physically handled automatically by making the constraint the content of a slot on the frame that represents the concept.

The expression for the constraint’s own head goal, needed by the constraints of other, ancestor concepts, can be derived automatically from the concept name and required arguments. The arguments represent the other values with which the owner concept’s value is related by the constraint. Because a single output value is defined (the output value for global data is handled automatically), all of the remaining arguments must be used as input arguments. The user can enter the list of variables required for the constraint header in a simple text editor.

The editor shows the guard and body of a constraint, and permits a user to edit each separately. A leaf node that takes on a constant value is treated as a special case. Each of these editors is entered from the procedure editor.

5.7 EDITING A CONSTANT VALUE

Concept nodes with no supporting nodes represent either constant values or values obtained interactively from the user. SITE possesses a relatively complete set
of interactive queries to obtain such values. This SITE module is reused in Designer. It provides for the creation of compiled constants, and menus to be presented to a user at run time. Menus in the constant value editor allow a user to enter individual values, choose a constant value from a list of values in a supporting concept, or specify that an end user, when the generated artifact is run to actually solve the problem, should select a value from such a list.

Designer automatically generates procedures for these nodes that assign the results of SITE knowledge acquisition procedures to Designer variables. These procedures drop out of the Designer computational structure. To reuse the SITE code for creation of constant values, they use Prolog data structures and code exactly as does SITE. This design choice made it more feasible to construct a usable first prototype of Designer. Future enhancements to Designer should change this functionality to be less like SITE and more like the rest of Designer.

5.8 EDITING A SUBGOAL BLOCK

The guard and body of a constraint are each a block of subgoals. The same editor is used for each. This editor permits a user to type a list of statements, each representing one subgoal, in a simple text editor. The statements are then parsed to check for correctness, and automatically transformed to a procedure-goal data structure as described in section 5.2.

The user can also specify a type for the block. Typing is currently only used to check that a guard block evaluates to Boolean.

Representing and inputting a constraint in text form implies the need for a language. Unique and distinct representations, or syntaxes, may be required by
different levels of abstraction. Two corresponding nodes, one high-level and one low-level, may represent the same level of abstraction, but in different languages. This is the reason for describing and compiling a constant value more simply, without specifying a complete constraint for it.

Most of Designer's editors exploit metaknowledge to present the user with lists of alternatives. For the subgoals of a constraint, it is cumbersome to follow this interface philosophy. A small input language is implemented for users to describe assignments and execution of procedures.

A guiding principle in a fully-realized design tool should be ease of use for the human designer. A more important criterion in the current Designer prototype is triviality of the language. This mitigates the problems of parsing and validating user input. The design language must account for the following procedural structures:

a. Evaluation of subgoals, and

b. Association of values resulting from subgoal evaluation with variables, to make them accessible to other subgoals or as a resulting value for the entire constraint and concept.

Each of these is described in the following paragraphs.

5.8.1 Evaluation of Subgoals

A constraint often relates the value of its owner concept to those of supporting concepts. To obtain the values of supporting concepts, it evaluates subgoals that
represent the constraints of those concepts. Any value used in the constraint
definition that is not obtained from a supporting concept must be obtained from a
built-in system primitive. These are also represented as constraints. Thus, every
subgoal in a constraint definition can use a consistent representation.

Designer presents constraints and subgoals to the user as procedural functions.
A familiar syntax is specified to represent each subgoal:

\[
\text{variable-name} := \text{constraint} \left(\text{variable-name} [, \text{variable-name}]^* \right)
\]

This treats each constraint subgoal as if it were a function call. As a function,
it returns a single value. This result value may be of some arbitrarily complex data
structure.

The experimental results reported in Chapter 6 suggest that this representation
should not have been used. It has the effect of leading users to mix declarative and
procedural styles. Possible improvements are discussed in section 8.5.

Designer provides some built-in, primitive system constraints (basically, the
ones provided by Prolog). These may be used as any other constraint names, except
that there need be no associated supporting node. These primitives include constraints
to manipulate the global variable list. The procedure editor will display for the user a
list of the available primitives.

Logical evaluation of a subgoal is implemented as a direct use of the subgoal’s
head as a Prolog subgoal. The formalization of the theory specified a uniform set of
primitive subgoals, including a ValueMap() constraint to return the value of a
subgoal. Pragmatically, the subgoal is used directly.
There is currently no provision for users to add definitions of their own built-in constraints. The system assumes that all user computations should have a structure expressed in the conceptual map, and that the provided built-ins are sufficient for the lowest level primitive computations. One topic for future work is an assessment of this assumption.

The operations within a constraint consist only of a list of subgoals. For an and-node, sequence is implied by the order of the list. The generated Prolog source code preserves this order. For an or-node, there is only a single subgoal. Designer assists the users to construct refining nodes such that they all share the same head goal, matching the subgoal of the parent node’s constraint. The value of the or-node constraint is then carried up directly from whichever of its refining nodes contains a constraint that evaluates to true. The alternate node definitions are evaluated in the order implied by the natural (i.e., Prolog factbase) order of the refinement relationships.

5.8.2 Use of Variables

Variables are recognized by their unique positions in a small number of syntax forms. Variables are implemented as Prolog logical variables. They can therefore be either unbound or bound to a value. A bound variable cannot be reassigned.

Designer uses variables in the following ways:

a. The head goal of a constraint uses variable names to represent each argument. These variables are available for use in other expressions within the constraint.
b. A subgoal evaluation binds (assigns, in the provided procedural syntax) the result of a constraint execution to a variable.

c. Each parameter in a constraint call must be either a variable or a constant value.

d. A subgoal may update the set of global parameters that are passed throughout the calling sequence. The global parameters are, therefore, both input and output, via different variable names.

The formal theory defines constraints in terms of values, and unbound variables as unknown values. In practice, Designer does not prevent unbound variables from being used as arguments to constraints. They have no logical meaning, but may have pragmatic significance to primitive Prolog predicates. An issue for future work is whether this should be restricted.

The return value variable and output global variable must be unbound on entry to a constraint.

5.9 CONCEPTUAL GRAPH EXAMPLES

The following examples show how Designer stores its internal representation for concepts and constraints representing sequential subgoals, choice, and iteration.
5.9.1 Sequential Subgoals

A single node from the example of Figure 1 follows:

((model doctrine-option))
((model operator))
((model applicable-doctrine-statement))
((supports doctrine-option applicable-doctrine-statement))
((supports operator applicable-doctrine-statement))

((BLACKBOARD-STRUCTURE (concept ?x applicable-doctrine-statement
  "applicable-doctrine-statement is a data structure."
  "It is constrained to be valid when a doctrine template"
  "matching the current doctrine-option is filled in with"
  "the user's selected value for operator.") and-node
(concept-procedure empty ?x g5 ()
  (procedure-block empty
    ((procedure-goal ?x true () globalIn globalOut guard))
    boolean)
  (procedure-block empty
    ((procedure-goal ?y doctrine-option () globalIn globalOut doctrine-option)
    (procedure-goal ?z template (doctrine-option) globalIn
      globalOut doctrine-template)
    (procedure-goal ?u operator () globalIn globalOut
      operator)
    (procedure-goal ?v insert-attribute (doctrine-template operator) globalIn globalOut)
5.9.2 Choice

The goal node from Figure 5 and its refining nodes to implement choice (not shown on that figure) follow:

```
((model c))
((model c1)
((model c2))
((refines c1 c))
((refines c2 c))

((BLACKBOARD-STRUCTURE (concept ?x10 c
  ("c demonstrates simple choice. Its value is true"
  "if either a or b is true.") or-node
(concept-procedure empty ?x14 g4 ()
(procedure-block empty
  ((procedure-goal ?x15 true () globalIn globalOut guard))
  boolean)
(procedure-block empty
  ((procedure-goal ?x16 c-choice () globalIn globalOut c))
  untyped)
  untyped ())))))
```
((BLACKBOARD-STRUCTURE (concept ?x10 c1
  ("c1 refines c") choice-node
  (concept-procedure empty ?x14 g4 ()
    (procedure-block empty
      ((procedure-goal ?x15 b () globalIn globalOut guard))
      boolean)
    (procedure-block empty
      ((procedure-goal ?x16 b () globalIn globalOut c))
      untyped)
  untyped ())))

((BLACKBOARD-STRUCTURE (concept ?x10 c2
  ("c2 refines c") choice-node
  (concept-procedure empty ?x14 g4 ()
    (procedure-block empty
      ((procedure-goal ?x15 a () globalIn globalOut guard))
      boolean)
    (procedure-block empty
      ((procedure-goal ?x16 a () globalIn globalOut c))
      untyped)
  untyped ())))

The Designer Software Tool 158
5.9.3 Iteration

The header node from Figure 6 and its refining nodes to implement iteration
(not shown on that figure) follow:

(((model iteration-recur))
((model iteration-base))
((model iteration))
((supports access-next iteration-base))
((supports access-next iteration-recur))
((supports operation iteration-recur))
((supports data iteration))
((refines iteration-recur iteration))
((refines iteration-base iteration))

(((BLACKBOARD-STRUCTURE (concept ?x37 iteration () header-node
    (concept-procedure empty ?x44 g13 ()
      (procedure-block empty
        ((procedure-goal ?x45 true () globalIn globalOut
          guard))
        boolean)
      (procedure-block empty
        ((procedure-goal ?x46 iteration-loop (input) globalIn
          globalOut output))
        untyped)
    untyped ()))))
((BLACKBOARD-STRUCTURE (concept ?x37 iteration-base
  ("iteration-base is base case for iteration."
  "Succeeds when data is empty.") and-node
  (concept-procedure empty ?x38 g9 ()
    (procedure-block empty
      ((procedure-goal ?x39 list-empty (input) globalIn globalOut
        guard))
      boolean)
    (procedure-block empty
      ((procedure-goal ?x40 = (input) globalIn globalOut
        output))
      untyped)
    untyped ())))

((BLACKBOARD-STRUCTURE (concept ?x37 iteration-recur () and-node
  (concept-procedure empty ?x44 g16 ()
    (procedure-block empty
      ((procedure-goal ?x45 true () globalIn globalOut
        guard))
      boolean)
    (procedure-block empty
      ((procedure-goal ?x46 access-next (input) globalIn globalOut
        pair)
      (procedure-goal ?x46 access-first (pair) globalIn globalOut
        head0)
      (procedure-goal ?x46 access-last (pair) globalIn globalOut
        tail0)
      (procedure-goal ?x46 operation (head0) globalIn globalOut

The Designer Software Tool 160
head)
(procedure-goal ?x46 iteration (tail0) globalIn globalOut
tail)
(procedure-goal ?x46 append (head tail) globalIn globalOut
output)
untyped)
untyped ())))

5.10 REFLECTING ON THE GRAPH

Reflection is examination of the conceptual graph to discover possible extensions to it. For a human user, this is an informal operation. The user creates extensions to the map by using any of the editors to add or delete concepts, relationships, and constraints. Designer’s automated evaluator agent is triggered by an explicit command in the prototype. It examines each feature (map, concept, relationship, constraint, and loop) in the design, matches it to relevant constraints in its design metaknowledge base, and creates data structures to represent the constraint violations it finds. Designer’s edit generator matches the constraint violations against its editing metaknowledge base, and creates editing commands to remedy the violations.

The evaluator is nothing but a constraint checker. It is only a little more specialized than Prolog itself, in having some knowledge about specific types of features to test. It could be extended to refer to specific knowledge bases for different purposes, or to allow extensions to its knowledge base.
Procedurally, the evaluator's operation is as follows:

initialize an empty list of constraint violations
for each constraint in the knowledge base
    access constraint as <applicable-feature, condition, body>
    look up features on blackboard whose type is applicable-feature
    if condition(feature) = true and body(feature) = true,
        append <constraint, feature> to list of constraint violations
    endif
endfor

Section 4.4.10 said that Designer needs metaknowledge about constraints and goals for conceptual maps, and about the connection between these constraints and editing operations. The practical operations actually implemented follow Stefik [42]. This is a beginning knowledge base. It needs to be validated against experience, and extended as Designer is used. The knowledge base is presented in Table 3.
<table>
<thead>
<tr>
<th>Feature Type</th>
<th>Constraint</th>
<th>Edit Operation to Resolve Constraint Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept Node</td>
<td>Every node without a constraint must be refined to a node that does have a constraint.</td>
<td>Create the refinement node. Create a constraint in the parent node to obtain the value of the child node. Create a constraint template for the child node with a single operation taking as arguments all supporting nodes.</td>
</tr>
<tr>
<td></td>
<td>Every node must have a value.</td>
<td>Create an output variable of the node’s constraint.</td>
</tr>
<tr>
<td>Cycles</td>
<td>Every cycle in the graph must be represented as a recursive constraint.</td>
<td>Choose a head node for the cycle. Construct a base node and recursive node. Interact with the designer to construct a base node guard predicate that defines when the recursion terminates.</td>
</tr>
<tr>
<td>Constraint</td>
<td>There must be exactly one assignment of a value to the output variable.</td>
<td>Create an assignment of the last intermediate computation.</td>
</tr>
<tr>
<td></td>
<td>Every supporting node must be used in the constraint of an and-node. (For or-nodes, all supporting nodes may be evaluated, but only one will supply a value.)</td>
<td>Delete the arc to the unused supporting node.</td>
</tr>
<tr>
<td></td>
<td>Sequential steps of a constraint must be represented as refining and-nodes.</td>
<td>Create new refinement nodes, and move the steps of the constraint into them.</td>
</tr>
<tr>
<td>Feature Type</td>
<td>Constraint</td>
<td>Edit Operation to Resolve Constraint Violation</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Concept Node</td>
<td>Every node without a constraint must be refined to a node that does have a constraint.</td>
<td>Create the refinement node. Create a constraint in the parent node to obtain the value of the child node. Create a constraint template for the child node with a single operation taking as arguments all supporting nodes.</td>
</tr>
<tr>
<td></td>
<td>Every and-node must be refined to or-nodes.</td>
<td>Create new refinement nodes (by default, one) for choice templates.</td>
</tr>
<tr>
<td></td>
<td>Every or-node must be refined to and-nodes.</td>
<td>Create new refinement nodes, and move the steps of the constraint into them.</td>
</tr>
<tr>
<td></td>
<td>Every called constraint must belong to a supporting node of the caller. (This also implies that nodes cannot call refinement nodes of other nodes: the refinement relationship is a tree.)</td>
<td>Add a supporting arc.</td>
</tr>
<tr>
<td>Graph</td>
<td>At least one leaf node must exist.</td>
<td>A violation means all nodes participate in cycles. Ask the user which point in each recursion will be refined to obtain data from knowledge bases or external sources. Create refinement nodes as needed.</td>
</tr>
<tr>
<td>Variable</td>
<td>Each input must be in the caller's input or local variables.</td>
<td>Remove the variable from the inputs to the called constraint.</td>
</tr>
<tr>
<td>Feature Type</td>
<td>Constraint</td>
<td>Edit Operation to Resolve Constraint Violation</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Concept Node</td>
<td>Every node without a constraint must be refined to a node that does have a constraint.</td>
<td>Create the refinement node. Create a constraint in the parent node to obtain the value of the child node. Create a constraint template for the child node with a single operation taking as arguments all supporting nodes.</td>
</tr>
<tr>
<td></td>
<td>Each variable must have exactly one assignment.</td>
<td>Create new variable names for later assignments. Delete unassigned variables.</td>
</tr>
<tr>
<td></td>
<td>Local variables used in a constraint call must have a previous assignment.</td>
<td>Remove the variable from the inputs to the called constraint.</td>
</tr>
</tbody>
</table>

5.11 CHANGING A GRAPH

Editing operations to change the conceptual graph may be obtained either from an interactive human user, or from the edit generator. After the evaluator finds constraint violations, the edit generator matches them to a knowledge base conceptually very like the second and third columns of Table 3. It sends the editing operation that accompanies a matching constraint to the appropriate editor.

Edit operations from the edit generator may have the effect of creating or deleting concepts or relationships. Some small changes can be made to constraints, but they do not go beyond constructing head goals.
Applying edits to a graph changes the graph. This implies that both the structure (syntax) and contents (semantics) of the design have changed. Any change at any level implies two types of ripple effects.

First, the physical structure of supporting nodes changes. Old nodes may have been deleted, leaving their former supporting nodes, which must be either connected to other nodes or deleted. New nodes may have been created. Each needs to be supported by the next lower level.

Second, the conceptual change may have altered the meaning of the concepts involved. It is possible that the supporting lower-level nodes no longer provide a valid support for the higher-level concept.

The first effect is syntactic, and the second semantic. A correct syntactic update to the map depends on a correct semantic update.

An important change found in software engineering is the change in a high-level requirement. While such a change is represented easily, as a change in the structure of the related nodes at one level, it is more difficult to represent the impact of this change on lower levels.

A lower-level concept is an argument for a higher-level concept. There are reasons and warrants for the claim that the one supports the other. This meta-level of support could be represented and manipulated in its own right. It has been, in Dasgupta’s Theory of Plausible Design [5]. That work includes a calculus for reasoning about the support for a design concept.
This thesis does not annotate a concept with the reasons it supports other concepts. The theory of conceptual maps and designs based on them has been kept very simple, to demonstrate a practical application of the theory. In Designer, a node supports a higher-level concept because that is assumed to be the intent of the designer.

A conceptual change requires the designer to consider the implications in levels to which the affected concepts are linked. Support by Designer for conceptual changes is limited to providing reminders to the designer, and ensuring that the structure of the linked nodes remains consistent. This is a syntactic check. Designer can detect a syntactic error introduced by a design change. It cannot determine what semantic change caused the syntactic change.

In a large design graph, the designer may lose track of detail. Designer does not yet address this problem. A special case is the problem of recognizing that two subparts of the conceptual graph actually represent the same meaning. If this could be automatically detected, Designer might automatically merge and check for consistency. This is a hard problem, and a possible topic for future work (see section 8.11).

Designer can save a conceptual graph as a file for later use. This has two important implications. First, it supports incremental development of a design. The work does not need to be done all in one session. Second, it supports maintenance at the design level. After the design has generated source code and a resulting computer program put into service, it is very likely that the system will need to be changed. A designer can retrieve the Designer conceptual graph for a system, re-enter the graph at any level of detail, and look for the source of problems or work on enhancements.
at the design level. This was one of the major goals discussed in Chapter 1, and an important reason to apply this work to software engineering projects.

It is possible to combine work done by more than one person at different times. However, Designer in its current prototype does not explicitly support this, and gives no assistance for checking that the partial designs from different files match up. This is a possible topic for future enhancement.

5.12 Generating Executable Code

Code generation is, conceptually, just another refinement to the design. The ability to generate Prolog source code easily and naturally from a design is additional evidence for the claim that analysis → design → code is a smooth transition of the same type.

Pragmatically, code generation is a significant problem, and very different from the type of extensions Designer does at higher levels. The automatic programming literature suggests that large amounts of specialized knowledge are needed to generate code automatically.

Designer postpones the commitment to specific language domain knowledge as long as possible. The design is refined to a low level of detail in the model of an annotated conceptual graph. Specification of the problem solution remains in a more abstract, thus computationally simpler, problem space until the subgoal definitions are a good match to the available knowledge about the target programming language.
The primary means for accomplishing this deferral of detail is the choice of Prolog as the target programming language. The theoretical computational model is constructed to map directly onto Prolog's computational model. Most of the difficulty of code generation is thus sidestepped. Designer's declarative representations are mapped directly onto Prolog's declarative representations. The constraint on each concept is mapped to a Prolog predicate.

It is interesting that the structural relationships are implied by the subgoals found in the constraints. The explicit relationships between concepts are not repeated in the generated Prolog code. Designer is thus seen to contain redundancy in order to more conveniently fit human design activities.

Generating code for Prolog uses only a small amount of domain knowledge as compared to traditional automatic program generators. The code generator is a simple Prolog predicate. Metaknowledge about Prolog contains a few trivial rules: nodes are mapped directly to clauses; local and input variables are mapped directly to Prolog variables; and output variables are mapped to an additional argument of the relevant predicate.

Designer does not explicitly use knowledge about the user's intended top-level goal, or entry point, in the conceptual graph. The evaluator constrains the map to have a single top-level node that supports no other node. This could be better used in a future version to supply the user with an intuitive startup command for the finished generated program. Currently, Designer iterates through the set of all concept nodes, generating code for each.

A working artifact must contain the set of primitive constraints that may be used as subgoals to the user's constraints. The Designer prototype includes these in a
standard library file to be consulted along with the generated source code. A future version would do better to automatically copy the ones actually used.

Designer recognizes that code can be generated when all design computations are fully constrained. It does not actually do so until the user requests it. Standard Edinburgh syntax is generated.

5.12.1 Generating Code for Sequential Subgoals

The subgoals in an and-node are translated directly to Prolog subgoals. The constraint names are used as functors. The following example shows the Prolog code generated from the example conceptual map structures of section 5.9.1:

```
'applicable-doctrine-statement'('globalIn', 'globalOut',
   'applicable-doctrine-statement') :-
   % Guard goals
   'true'('globalIn', 'globalOut', 'guard'),
   !,
   % Body goals
   'doctrine-option'('globalIn', 'globalOut', 'doctrine-option'),
   'template'('doctrine-option', 'globalIn', 'globalOut',
     'doctrine-template'),
   'operator'('globalIn', 'globalOut', 'operator'),
   'insert-attribute'('doctrine-template', 'operator', 'globalIn',
     'globalOut', 'applicable-doctrine-statement').
```
5.12.2 Generating Code for Choices

Choices are already structured in the map as a set of alternative nodes, directly analogous to the set of alternative clauses in a predicate. The following example shows the Prolog code generated from the example conceptual map structures of section 5.9.2:

'c'(_globalIn', _globalOut', _c') :-
  % Guard goals
  'true'(_globalIn', _globalOut', _guard'),
  !,
  % Body goals
  'c-choice'(_globalIn', _globalOut', _c').

% c1
'c-choice'(_globalIn', _globalOut', _c') :-
  % Guard goals
  'b'(_globalIn', _globalOut', _guard'),
  !,
  % Body goals
  'b'(_globalIn', _globalOut', _c').
% c2
 'c-choice('globalIn', 'globalOut', '_c') :-

 % Guard goals
 'a'('globalIn', 'globalOut', 'guard'),
 !,
 % Body goals
 'a'('globalIn', 'globalOut', '_c').

5.12.3 Generating Code for Iteration

Iteration is already structured in the and/or graph as a base case and recursive case. A Prolog base clause is generated first, to stop the recursion when its guard is satisfied. A recursive clause, which has the next node in the iterative loop as a subgoal whenever the base clause is not matched, is also generated. The following example shows Prolog code for the example conceptual map structures of section 5.9.3. Code that was not correctly generated because of a problem in the prototype has been extrapolated to complete this example.

'iteration('input', 'globalIn', 'globalOut', 'output') :-

 % Guard goals
 'true('globalIn', 'globalOut', 'guard'),
 !,
 % Body goals
 'iteration-loop('input', 'globalIn', 'globalOut', 'output').
% iteration-base
iteration-loop(_:input', '_globalIn', '_globalOut', '_output') :-
  % Guard goals
  'list-empty'('_input', '_globalIn', '_globalOut', 'guard'), !,
  % Body goals
  '='('_input', '_globalIn', '_globalOut', '_output').

% iteration-recur
iteration-loop('_input', '_globalIn', '_globalOut', '_output') :-
  % Guard goals
  'true'('_globalIn', '_globalOut', 'guard'), !,
  % Body goals
  'access-next'('_input', '_globalIn', '_globalOut', '_pair'),
  'access-first'('_pair', '_globalIn', '_globalOut', '_head0'),
  'access-last'('_pair', '_globalIn', '_globalOut', '_tail'),
  'operation'('_head0', '_globalIn', '_globalOut', '_head'),
  'iteration'(_tail0, '_globalIn', '_globalOut', '_tail'),
  'append'('_head', '_tail', '_globalIn', '_globalOut', '_output').
5.12.4 Generating Code for Other Target Languages

Prolog encourages nondeterministic programs. Designer permits, but discourages it. Each or-node, when executed, takes advantage of nondeterministic backtracking to search for its first successful alternative. The generated code uses the Prolog cut operator ("!") to make each constraint deterministic after a guard succeeds. This permits simpler consideration of the constraints as units.

It also prepares Designer for future extensions to generate languages other than Prolog. It would be desirable to generate code in other languages considered more standard by various professional disciplines for different purposes. It is difficult to automatically translate Prolog with backtracking to most procedural languages.

Designer solves this problem by making each generated predicate deterministic. Backtracking within the guards is a much smaller problem than supporting general backtracking at any point. Guards of alternative constraint definitions can be easily translated to conditional switch or case structures.

When this work is extended to other implementation languages, it will probably be advantageous to treat Prolog as another level of abstraction between the conceptual map and a procedural language. This would also provide another bit of evidence to support the claim above that there is no difference in kind between design and code: the target code of a first version may become just another design level in a later version.
5.13 INTELLIGENT FEATURES OF DESIGNER

The primary goal of the Designer tool is to use the Computational Theory of Design to support actual design. It is intended to be used in the field, for real software problems. Therefore, demonstrating a high level of intelligence in the tool has not been a priority. Rather, the primary insights about intelligent behavior are found in the theoretical model of how designers use concepts and conceptual tools. The intelligent characteristics of Designer itself exist for a practical reason: they make it more powerful at supporting design.

The primary intelligent features of Designer are as follows:

a. Use of a conceptual model to drive knowledge representation and software architecture.

b. Imitation of the human cycle of action, reflection, and extension to a design domain theory.

c. Automatic search for opportunities to refine concept nodes and to create new parts of the design.

d. Use of domain knowledge concerning how design is done to create templates for new parts of a design.

e. Cooperative design. A human agent and an automated agent coordinate their search for opportunities to bring the same object closer to shared goals.
Most of the intelligence of the Designer tool is contained in its ability to find and fill in the middle layers automatically. It searches opportunistically for nodes in the map that need to be extended to lower levels, or that need to be connected to higher levels. It uses compiled plans to generate proposed nodes from templates that fill in the middle layer, and then supports the designer in editing the proposed solution.

Designer does not generate an original design. All of its semantic knowledge of a specific design is contained in the structure of relationships and constraints given by the user. It does not contain knowledge bases to move semantically closer to the user’s goal.

It can, however, find incompleteness in a design, and proposes extensions to make those situations more complete. A semantic meaning needs to be supported by a syntax or representation. Designer contains general metaknowledge about the structure and syntactic requirements of conceptual maps. It can automatically move a map syntactically closer to one with sufficient detail to support a semantic goal.
CHAPTER 6
EMPIRICAL EVALUATION OF CONCEPTUAL MAPS

Several experiments were performed to evaluate the usefulness of conceptual maps and constraint annotations. Designers working with actual software projects were given the prototype Designer program. The experimental users were asked to answer questions about working with conceptual maps, with designs, and only secondarily with the Designer prototype. Designer as presently implemented is expected only to demonstrate the value and feasibility of conceptual maps and the Computational Theory of Design.

The results give empirical evidence that the conceptual map is a useful tool for software engineering, and that the Computational Theory of Design describes the design process accurately. The central metaphors of messing about with a problem and mapping the concepts in the problem were easily understood and used by people with no prior exposure to conceptual maps, including one complete novice to software modelling.

This chapter describes the experiments and their results.

6.1 EXPERIMENTAL METHODOLOGY

A person from each of four different actual projects was approached for assistance with the Designer experiments. A future version of Designer is expected to be made available to all these projects. The experiment was designed as an opportunity for them to see and use the first prototype of a new software tool. The information and instructions given to the experimental users stressed that information
collected from them will be used to write problem reports against Designer. Within
CSC's contract with the U.S. Navy, problem reports are the normal mechanism for
defining future changes to a baselined program.

6.1.1 Experimental Protocol

A standard set of tasks was developed for each experimental user to perform
with Designer. Each user worked from the same written introduction and protocol.
The protocol is included as Appendix A. It asks each user to supply actual design
information from his/her own ongoing software development project.

Each user was present at an oral introduction to the tool and conceptual
mapping. In each experimental session, one person from one project worked with the
Designer tool for at least 1.5 hours. An experimenter sat with the user to answer
questions and provide assistance if needed. The written tasks were used as the
primary source of information during the experiment. The experimenter did not offer
any information unless asked by the user. The experimenter took notes on the actions
performed and comments expressed by the user. Electronic recording devices were
not used because of NSWCDD security regulations. After the session, the user was
given a standard questionnaire of 12 questions. The questionnaire was returned to the
experimenter later.
6.1.2 User Profiles

The experimental users were chosen for the following reasons:

User 1 is a professional statistician, with 30 years' experience writing mathematical software. The user is currently working on a program to predict trends in software quality from metrics of historical baselines. This project was used for the experiment. The user has never before worked on a project that used requirements or design. The experimenter hoped to find how well Designer would work in this problem domain (to which Designer may actually be applied in the future), and how easily a user with no exposure to CASE-type tools could learn its concepts.

User 2 is a professional software engineer with experience in robotics and other AI areas. The user is currently working on a large persistent knowledge base project. For use in the experiment, this user chose a robot vision problem (edge recognition). The user has an expert knowledge of the market, abilities, and limits of software support tools. The experimenter hoped to get comprehensive feedback from a user known to have very high expectations for software systems.

User 3 is a manager with software engineering experience. The user is responsible for several different software projects. This session was NSWCDD/N86's opportunity to evaluate the product as delivered under CSC's contract tasking. The customer preferred to use the session as a demonstration rather than as an experiment, using the same AEGIS example presented in section 4.3.2.

User 4 is a software engineer who specializes in human-computer-interfaces. The user is currently working on simulations for Navy training. For use in the
experiment, the user chose a variant of the Travelling Salesman problem. The experimenter hoped to get experienced reactions to the HCI from this user.

6.1.3 Data Collection

The following three types of data were collected:

a. The experimenter's notes on problems encountered and significant comments or questions by the users. Some of this information is discussed below in section 6.2. Some of it is included in the Designer problem reports, presented in Appendix C. This is qualitative data that depends on subjective user reactions, but can be tracked to specific technical features in the program.

b. The users' responses to a standard questionnaire. The questionnaire is presented in Appendix B.1, and the responses in Appendix B.2. For all of the questions on the survey, 1 was the least favorable response and 5 the most favorable response. They are qualitative evaluations by users and can be used to judge design decisions and to guide future enhancements.

c. The conceptual map files and Prolog source code files generated by Designer. These were examined for evidence of program bugs. Items found in this examination are reflected in the Designer problem reports, presented in Appendix C.
6.2 RESULTS

All users understood conceptual maps easily, could work through real design problems with them. All gave good ratings to the operations concept and basic principles shown in the Designer prototype. All of them had some difficulties using the Designer prototype tool.

This section presents the empirical findings of the experiments, and discusses issues they raise for future use of the Designer tool.

6.2.1 Qualitative User Responses

Users strongly agreed that conceptual maps are a useful way to do design. This response received the highest ranking found in the survey. Its mean score was 4.5, with a standard deviation of 0.5. The standard deviation tied for the third-lowest standard deviation for the survey responses. This indicates a high level of agreement among the users on the high score. The standard deviations, interpreted as level of agreement on the mean score, ranged from 0.43 to 1.22.

The overall reaction to the Designer tool was a mean score of 4.0, with a standard deviation of 1.22. This suggests that the users thought Designer has promise as a tool, but there was less agreement on the score. Other questions dealing with the effectiveness of the current prototype received lower scores. However, the only topic with a mean score below 3 (the midpoint of the scoring range) was "Error messages helpful." Table 4 shows the complete results.
User 1 commented "The tool forces you to think logically. That's very good. Without it you'd make assumptions that you were thinking straight on things like relationships and flows." This user succeeded at putting part of an actual project design into Designer, then discovered that some assumptions in the project were not correct. The comment "I need another support coming in here" was followed by an update to the user’s own design, resulting from reflection on the conceptual map. This experiment thus gave some immediate benefit to the user’s work.

This user had difficulty using the syntax of constraint subgoals to annotate a concept, and encountered several software errors in the system. Despite this, the user could explore the design fragment, and look at several different ways to organize it. The other users had similar experiences. Their comments agreed that the conceptual map helped them to organize their problem settings and explore their partial designs.

These experiments did not examine the effect of training on user's success with and attitudes toward the tool. The informal training sessions were no longer than 15 minutes each, supplemented by assistance from the experimenter as needed during the experiment. No training materials were prepared. The central topics were difficult to present without explanatory pictures, so most of the training was done by showing the tool's editors.

6.2.2 Technical Evaluation

Several technical shortcomings were found. The primary technical objective of the experiments (that is, beyond the qualitative user evaluation) was to discover software errors within Designer. Because it is a prototype, only cursory measurements were made of program efficiency.
Designer has been installed and used on a VAX 6460, a SUN SparcStation 2, a SUN SparcServer 670, and an Amiga with an MC68000 CPU. It has not yet been run on an Intel CPU. Although the VAX contains six high-performance CPUs and supports hundreds of users, Designer ran too slowly on it to be useful. Performance was adequate on both SUN platforms. Designer ran too slowly on the Amiga for any purpose other than development. However, it ran only slightly slower there than on the VAX. An Amiga (or Macintosh) with a 68040 CPU would give better, probably adequate, performance. Speed did not seem to have any relationship to the size of the conceptual map.

Despite its slow running speed, productivity with the prototype is high enough to be useful. A conceptual map of six nodes can be built in a session of one to two hours. From this, Designer generates several hundred lines of Prolog source code. That compares favorably with the time typically required to write the same code by hand. It compares unfavorably with other CASE tools that generate code. For example, CSC personnel have built design models of 15 object classes in the tool Ptech in a three-hour session. The C++ code generated by Ptech from these models contain tens of thousands of lines of code.

Designer also consumes excessive memory. For a conceptual map with only 6 nodes, the entire Designer blackboard and associated Prolog facts (when written to a Prolog source file) is 2 kbytes. The Designer source code is 500 kbytes, and a VPI Prolog environment file for the system is 400 kbytes. The heap was believed to contain, at most, one copy of the Designer blackboard. Despite these small numbers, Prolog heap usage varied between 1.4 Mbytes and 2.0 Mbytes. Heap usage increases with the size of the conceptual map, at a rate of approximately 300 kbytes per node.
The memory usage limits Designer's usefulness on smaller platforms. Even on the SUN SparcServer, experiments have not yet determined the memory required for a large conceptual map. The most common cause of memory problems in Prolog is its recursive backtracking nature. However, these problems would show up in the trail and variable stacks. Heap is associated with the factbase.

The causes for these resource usage problems have not been analyzed. Possible solutions range from recoding Designer in a non-object-oriented style, to hosting it in a different version of Prolog, to redefining its required deployment environment.

During the experiments, 21 new problem reports were written against the Designer code. These should be resolved at the same time as any enhancements to create the next version of the program. A summary of the problem reports is found in Appendix C.

6.3 CONCLUSIONS

There may be two obstacles on the path to user acceptance: usefulness of the underlying metaphor, and usableness of the finished tool. Designer scores high on the first, but the current prototype scores low on the second.

The balance between usefulness and usableness appears subjectively different to different users. Two of the users were very familiar with existing CASE tools. User 4 rated Designer's effectiveness compared to other tools as 4, and said "I've seen other design tools, and I'm not impressed by them. I think it's more intelligent - a better way to go." User 2 rated this topic as 1 (the lowest score), and said
"Compared to other CASE tools, it's unusable." Significantly, User 2 also commented "The idea is good, though."

All the users approached Designer in terms of what they already knew. The conceptual map is a new way to think about problem structure, but all users connected it to other ways to think about structure. Many comments compared conceptual maps to structure charts and data flow diagrams. All users also grasped easily the notion of a concept as the central building block of design. The graph language, and the representation of concepts as nodes, seemed intuitive to each user.

It was much more difficult for non-Prolog programmers to understand and use the Prolog-like computational model. Annotating the map with constraints was the most difficult step for each user. All felt that using the problem structure to express control information was counterintuitive. Even after going through multiple examples of setting up an "if-then" choice as an operation on two supporting nodes, one user still automatically wrote traditional "if-then" syntax in the body of the parent node. Users with a knowledge of and/or trees understood the mechanism, but still found it difficult to use. No user constructed a correct loop structure. After giving examples and advice, the experimenter still had to step in and build each loop that was wanted by a user. In each case, the user was surprised to see the additional nodes and structure used to represent a loop, and decided to take the loop out of the experiment. All users radically simplified the original problems they had planned to use in the experiment.

All users wanted to write traditional imperative pseudocode within the body of constraints. Designer may have encouraged this by using a syntax intended to be easier for users to recognize and use. A constraint uses only applicative statements, but they resemble assignment statements. However, users did not perceive the
procedural style as familiar. Although the syntax was based on C, one user commented "Only real problem was the forced use of the FP language, instead of a more common syntax like C or Ada." The effort to rephrase the theoretical use of declarative knowledge in a procedural form was an error.

The experiments suggest that Designer can be developed into a useful tool. Its problem structure model is accepted by users, and can be used effectively in the tool in its prototype form. However, Designer's computational model is very different from users' prior knowledge and expectations. Users did not accept Designer's way of expressing constraints, algorithms, and even primitive data. This is a hurdle for users that must be removed before the tool is released to a production environment.

The problems with the constraint syntax may have been due to mixing the declarative and procedural paradigms. A more natural syntax for constraints as declarative descriptions would encourage users to continue describing the problem, rather than suddenly switching over to prescribing how to implement the constraint.

Designer's Human-Computer Interface (HCI) makes several incorrect assumptions and mixes the procedural and declarative metaphors in a way that is not helpful to users. Simpler metaphors are required in the HCI. Users expected high-level control constructs to be summarized in a single concept. This is reasonable, since control abstractions are, themselves, concepts. They are also taken for granted and used intuitively by software engineers. Designer's handling of choice and iteration is much more primitive, and requires the user to specify each step in each case. It should support these features at a much higher level of abstraction.

All of the users built a simple map at a single level of abstraction. In the text interface, the two ways to link concepts are syntactically similar. The practical
implications of using support and refinement together needs to be revisited in the next version of Designer.

Because a graph is an intuitively graphical object, it is highly desirable to build a graphical graph editor. Given the graphical environment, it would be desirable to find consistent ways to represent other aspects of the map and constraints in a declarative graphical syntax. This would also promote on-line help and tutorial explanations of the map and its features. User 2 commented "A graphical user interface and graphical program writer (icons) would solve the cosmetics and entice the user more than simple text interface."

An iconic language may be both more usable and a simpler semantic match to the way humans express problems. An iconic representation of concepts in the design bears a surface resemblance to a declaration of the concepts, and may be an appropriate way to express declarative constraints. No research was done during this project into the literature of iconic representations. Some suggestions toward this end are included in section 8.5.

The experimental results show little about the ease of using declarative annotations to the graphs. The prototype led users to think procedurally about the constraint annotations. In addition to the improvements discussed here to the user syntax for constraints, future work should explicitly try to discover if users find declarative annotations a natural and useful design technique.

In summary, the experimental record gives encouragement for pursuing Designer as a tool for eventual production use. However, the experiments have also produced indications for changes that should be made before Designer is applied to large projects.
Designer was originally planned as a VAX-based program. It is clear that it is not suitable for that environment in its present form. The desire to move to a graphical user interface suggests that a non-VAX platform would be more suitable for the next version of the tool.
CHAPTER 7
RELEVANCE TO SOFTWARE ENGINEERING

The motivation for this research was to apply some contributions of AI to the problems associated with software engineering. A basic goal of software engineering is to develop a scientific grounding and framework for software development. The Computational Theory of Design provides such a framework. A second goal is to lift the work of software development is done from the coding level to the design level. One idealistic expression of this goal is "to make coding obsolete." This is exactly the purpose of Designer.

The theoretical contributions of this thesis are one step toward a consistent, formal understanding of what is actually done when software developers build systems. The Designer tool is an attempt to develop software at the design level, and to generate code automatically from a design. Earlier attempts to develop software at the design level concentrate on automatic programming from algorithms. The problems and issues raised by that work are addressed by Designer, not by extending the model of how to code, but by replacing that model with a much simpler one. The difficult problems of automation may be solved more easily at the design level. Design solutions can then be fed to much simpler code generators.

The single design framework can be used to support all design activities: requirements specification, system design, and automatic code generation. This consistent, top-to-bottom support suggests a practical realization of the long-sought goal of moving software maintenance from code to the design level.

This consistency is the major reason that this design model can be implemented as a practical, automated system for design support. Design goes
beyond current CASE tools, in that it discovers refinements that can be made to a
design.

This work gives some reason to suspect that design and problem-solving may be
equivalent. The essence of design is to plan for change in the world. Any effort
to solve any problem implies just such a change.

This speculation is connected to the proof that the Computational Theory of
Design is Turing-complete. Because any computational approach to problem solving
can be expressed in a Turing Machine, any general design can also be expressed
within this design model. This suggests that design is a fundamental paradigm of
problem solving. Planning, heuristic classification, and other AI topics might be
restated as special cases of the design problem.

This report does not attempt to develop the support needed to prove such a
strong claim. The possibility is intriguing, and is an appropriate topic for further
research. If design were to be taken as a fundamental paradigm for computational
computer science, other practical issues must be considered.

Especially interesting is the question of how to determine which problems are
appropriately cast in this computational structure, and which are more easily solved
within other paradigms. This theoretical result also suggests that other practical
design models, such as Data Flow Diagrams, could be expressed as annotated
conceptual maps. Verification of this suggestion would require effort to map the
models onto each other, and probably additional work to make the translation a
practical and useful thing to do. A topic for future research is whether parts of
designs could and should be automatically transformed between multiple
representations for the convenience of users at different times in the design process.
7.1 REUSABILITY

Software reuse has been studied for years. The progress of projects such as STARS [41] has been limited by several issues. One important outstanding problem is how to represent the description of a piece of software so that it can be matched to future needs.

One advantage of representing a system by a conceptual map is that it lacks the complexity of lower-level code. The theoretical work of this report identifies sections of a design by high-level conceptual descriptions. Annotated conceptual maps may be more appropriate entities to store in a reuse repository than other design representations. The ability to index on concepts could be exploited to create a library of searchable and reusable designs.

Empirical experiments in this area would be useful. Subgraphs of a conceptual map can be characterized by some of the same criteria used in structured design methodologies: cohesion within a subgraph and coupling between subgraphs. Eden suggests that these measures are good indicators of the central concepts in a map. It might be possible to use existing information retrieval strategies, and to obtain better retrieval results from conceptual maps than from code libraries.
7.2 METHODOLOGICAL USE

This report presents a tool and an automated method, but not a methodology. It will be best judged after real experiences to design real systems. It is necessary to learn empirically how well Designer works, how it might fit into a methodology, how it works with, and how it compares to the traditional software methodologies.

All the common methodologies claim that a precise statement of the requirements of a software system is necessary. A requirements document is the criterion for recognizing an adequate and finished product, the blueprint and rationale for proceeding with design, and the skeleton for test plans. The assertion of this report that formal requirements are mythical (discussed in section 4.2.2) will probably be a major impediment to acceptance of this design theory by software engineers.

This makes it important to show how a traditional requirements document can be derived from the design style discussed herein. The document can then be a familiar mental framework and view for a project. Similarly, the structure of the network of concepts should be translatable to a conventional structure chart.

Modern practice emphasizes decomposition into modules. Designer uses a flatter approach: there are many more "modules" at the top level, if each node is considered a module. There is a "top-level module": the one producing a final output value. This is supported both horizontally and vertically. It is not easy to tell, as with a structure chart, which subset of nodes forms the support for a single higher node.

The conceptual map is not a new approach. Eden's work was published in 1983 [7], and builds on work about System Dynamics first published in 1961 [10]. It

Relevance to Software Engineering
is, however, unfamiliar to the software engineering community. It may be difficult to combine with traditional structured techniques. A solid study of real applications might suggest the most appropriate way to introduce this advance to a traditional software development environment.

Acceptance will require a mapping of the Designer paradigm onto current methodologies. The "distinctions that are not distinctions" are helpful here: the top level of a conceptual map represents high level analysis, the middle, constrained nodes represent traditional design, and the lowest level represents coding. Traditional requirements are buried somewhere in the constraints, below the problem setting level.

An important future task will be to discover how to derive traditional software engineering documents from a Designer conceptual map. A derived requirements document would be a familiar mental framework and view for a project. Similarly, the structure of the network of concepts should be translatable to a conventional structure chart. These are topics for future work, and are addressed in section 8.8.

An important change in software engineering since about 1980 is the use of CASE tools. Recent CASE tools have introduced novel design models to supplement the traditional data flow diagram for analysis and the structure chart for design. Examples include activity models, object models, process models, and object lifecycle models. Professional practice now recognizes that multiple views onto a complex system are necessary. This attitude should make it easier to introduce Designer as a new view of design.

Although Designer would replace some functions of other CASE tools, it would not be a monolithic solution to all of the problems of software engineering.
Other tools would still be needed. Examples include graphical tools to manipulate traditional views (such as data flow diagrams or structure charts) generated by Designer, simulators, requirements analyzers, and report generators. Designer could export information to these other CASE tools. Designer should adopt emerging industry standards for knowledge interchange to fit into a software engineering environment that may include multiple commercial products as a "plug and play module." See section 8.8 for further discussion of how future versions of Designer might fit into current software engineering practice.

Currently, Designer generates Prolog code. This is because Prolog is a good match to the simple and/or computational model of Designer. However, very few commercial software systems are coded in Prolog. A more mature Designer would use the same framework to generate code in other (probably procedural) languages.

7.3 SOFTWARE MAINTENANCE AND METRICS

One phase of software development to which Designer might be readily introduced is maintenance. Most debugging and correction activities are completely manual. This is at least partly because design information is rarely maintained in a manner useful to software maintainers. Maintenance could be more productive by finding problems in a conceptual map, and regenerating code after making corrections at the design level. Given the large proportion of software engineering budgets devoted to maintenance [44], this is an important result.

This comparison could be done by collecting the same metrics that are gathered during a traditional software project. Comparison of measures of effort and
reliability under the two environments could give a quantitative measure of the benefits to be gained.

A topic not yet examined is how to help maintainers find the correct level of abstraction, and the correct location within a large graph relevant to a symptom. Further work might find it appropriate to use encapsulation and information hiding techniques.

Once code is reliably generated from design, it no longer makes sense to debug and maintain code. Maintenance can move to the design level. Bug reports can be treated as design problems. Maintainers can work more productively with the conceptual map, instead of hunting bugs in source code. Code can be regenerated when the problem has been fixed in the conceptual map. The code automatically stays in synchronization with the design.
CHAPTER 8
FUTURE WORK

Most of the work of this thesis focused on developing the Computational Theory of Design. This chapter discusses extensions and additional applications of the theory.

The Designer software program is intended to demonstrate that the theory gives a sound basis for practical software development. However, the Designer program is a proof of concept. It is not a polished system, and is not sufficiently robust to support a professional software engineering environment. This chapter discusses how Designer could be improved to support actual software development projects.

8.1 "WHAT-IF" ANALYSIS OF DESIGNS

NSWCDD/N86 wants this work and the Designer tool to support "what-if" analysis of AEGIS designs. "What-if analysis" suggests (to use Eden's term [7]) messing about with design ideas, changing portions of a design, and then predicting or deriving the implications of the change. Its purpose is to compare alternative design solutions and to find the best design decisions.

Prediction of properties resulting from a design change might be interpreted to mean several different things. A future task for "what-if" analysis will need to choose and identify one of the following interpretations:
8.1.1 Evaluating Design Alternatives

Comparing multiple proposed partial answers to design problems is closely related to Simon’s discussion of assigning scores to partial paths of a search through a space of problem solutions [39]. Simon would like to predict the "goodness" of a design alternative before spending effort to work through it. Such an operation would require solid metrics attached to metaknowledge about design choices. The appropriate metrics to use for this purpose are not yet known. The discovery of good metrics and metaknowledge will require human analysis of sample design alternatives. The first sets of "what-if" experiments should create pairs of two design alternatives, and discover what attributes make one better than the other.

Simon’s model of a design space with search paths suggests that to create two design alternatives, Designer would follow two search paths all the way to the finished designs at their ends. But most design choices affect only a small part of a design. It is much more effective to create partial designs, and analyze what attributes make one a better answer to some specific design question.

Simon’s approach to design is modified by the Computational Theory of Design to become an editing problem rather than a search problem. Designer follows Schön, who models a search for a problem solution as an extended, iterative sequence of small changes to a partial solution. A partial change to a design is made by altering the value of one of these attributes. A single change creates a pair of alternate designs whose differences are easy to recognize and evaluate.

This model suggests that large parts of an existing design remain the same when a designer changes it. "What-if" experiments can be supported by constructing a new partial design containing one, or a small number, of changes. Useful
comparisons are then easy to compute. Humans can use the resulting analysis to create metaknowledge about how to evaluate the change as desirable or undesirable.

The architecture of Designer is amenable to storing and comparing two maps that are almost the same. It is easy to share nodes and other features between multiple conceptual maps. Currently, Designer neither identifies multiple maps, nor understands the meaning of two instances of nodes being versions of each other. It will need these extensions to support "what-if" analysis.

8.1.2 Change Detection

A design change made during an exploration of alternatives is likely to be small. The simplest analysis of such a change is a report on the differences between the original and new design. If the Designer program could detect changes, it would be easy to make a change detection report for human analysts. This enhancement would be much smaller and easier than adding knowledge about the significance of changes.

The possibility of applying graph theory to a conceptual map is discussed in section 8.11. The smallest and simplest application would be an application to find the differences between two graphs, to generate a change detection report. Such an application would identify unchanged nodes as anchor points of the two design alternatives, and methodically explore the portions of the map that did not correspond exactly.

A simple graph traversal program should be very reliable for small changes. Larger changes would introduce synchronization problems analogous to those known in programs that detect differences in blocks of text.
8.1.3 Generation of Standard Attributes

The purpose of experimenting with design and generating alternatives is to discover one alternative that is better than others. The meaning of "better" is often informal and approximate when used by human analysts. If a formal and rigorous definition can be developed, an automated application could apply this definition to the changed part of a design, and compare its goodness to the original design.

A formal definition of "good" for any concrete object, such as a design, is that it possesses desirable properties. A desirable property is an attribute that describes the object, whose value falls within a defined sub-range of the domain of possible values for the attribute. Appropriate attributes and values are defined by human designers. To be meaningful, the designers must define constraints by which the values of the attributes can be computed from other available information about the design.

Desirable design properties can be modelled as attributes analogous to the computed value of a concept node. The Designer software could contain definitions of design attributes in its design metaknowledge. The design metaknowledge would also include rules to compare the values computed for two versions of a design. The comparison rules would decide if one set of attribute values is better than another.

Design values can then be computed in the same way as the concept values are computed for a conceptual map. The procedure that describes how to compute each design property will use, as input, values computed for nodes of the conceptual map. A higher-level procedure would compute design attributes for each version of the
design, supply those results to the comparison rules, and generate an evaluation of the two design versions.

A first version would generate values for standard attributes (e.g., design complexity or memory usage). This can be taken a step further, to describe desirable properties unique to a specific design. This knowledge might be modelled either as attributes of existing concept nodes, or as a new type of node in a conceptual map. Values of such properties could then be computed for all versions of a specific design, independently of those attributes that describe all possible designs.

These options (general properties described in the system's metaknowledge, and specific properties stored in a conceptual map) are not mutually exclusive. Both general and specific properties of design are likely to be useful. General design metaknowledge could be encoded as a knowledge base of the Designer program, while that metaknowledge specific to each design is encoded in the conceptual map for that design.

In the same way, knowledge about how to compare the properties of two alternate designs can be stored either in the system or in the conceptual maps for specific designs.

The closeness of a conceptual graph to a tree suggests that some analysis questions can be answered without the expense of computing the top-level design value. Design metaknowledge could be applied locally to a subset of concept nodes. The paradigm of editing a design suggests that most design changes affect a small part of the complete design. The effects of small changes can be compared locally, and answers about a small group of concepts computed by evaluating only those concepts.

Future Work
8.1.4 Compute Results Beyond the Map

Originally, N86 suggested a set of specific questions to be answered for specific types of "what-if" scenarios. One example was "If a procedure is removed, can any data be removed?" Each of these is a mini-application, asking questions about the system that is larger than the scope of the local properties discussed in section 8.1.3. Each such application would give a designer advice on one specific topic.

In principle, answering questions of this sort is different only in scope, not in kind, from ranking one of two alternate designs as better and the other worse, as discussed in section 8.1.3. It would require broader metaknowledge, including rules to derive each desired conclusion. This metaknowledge would reside in an application completely separate from any conceptual map. One or more maps would be used as input to the application.

This section has identified three possible ways to use the Designer program to perform "what-if" analysis on AEGIS designs. The three were discussed in increasing order of difficulty: each application would require the abilities of the preceding ones. All three would require additional enhancements and extensions to the system described in this report.

8.2 EXPERIMENTS WITH DESIGNS

If one or more of the "what-if" analysis projects described in section 8.1 are not begun immediately, the Designer program will require additional exercise on

Future Work
actual design projects. The results reported herein are drawn from a small amount of experience. This report extrapolates many conjectures and hopes from those few results. However, these extrapolations can be readily tested.

Future projects should develop designs of real systems in Designer, and compare them to existing work on those same systems with older development tools. Useful metrics would include the effort required to produce code, error rates in generated code, and the effort required to make changes and to fix errors.

Section 4.5 contains a proof of the Turing-equivalence of the augmented and/or graph. The claim that any practical problem can be represented in this scheme is less formal and less certain. It must be validated by experience with actual problems.

8.3 INTERFACES TO OTHER TOOLS

The Designer software assumes that all design information is input by an interactive human designer. Much of the work done for N86 involves automatic extraction of design features from source code. The "what-if" experiments discussed in section 8.1 will mainly explore changes to existing code. These activities include both reverse engineering and maintenance of problems discovered in running systems.

It is neither desirable nor (for large systems such as AEGIS) efficient for humans to re-enter design information already found in existing code interactively. The Designer program should therefore automatically accept design information created by other tools. Besides using resources better, automatic input of existing data eliminates data entry errors.
It should be possible to construct the first draft of a conceptual map automatically from an existing system. Software objects are obvious candidates for low-level concepts, and relationships between software objects can be directly modelled as conceptual links. The resulting map will probably be appropriate to the lower levels of refinement. A human designer will probably add higher levels of abstraction to the automatically-constructed map.

8.4 OTHER USES OF CONCEPTUAL MAPS

Conceptual maps were not initially developed for software. The design theory developed here is applicable to general problem solving. Dasgupta makes this point strongly, and applies the theory in his own work mostly to the design of computer hardware. This report is about its application to software engineering. Future projects could usefully explore the application of design theory, and Designer, to other problem domains. Examples include simulation and planning.

8.5 DESIGN IMPROVEMENTS

The experimental results discussed in section 6.2 suggest that software engineers find the Prolog-like computational model radically different from their expectations. They prefer (and therefore need) an interface to Designer that uses more familiar models and metaphors.

Two problems arise in the descriptions of constraint annotations to the conceptual map. First, this part of the tool should use a declarative model. Constraints should use an annotation language rather than a coding language.
Second, Designer needs a richer vocabulary of abstract control and data constructs. Users must currently use low-level tools and primitive constraints to represent processes and data. There is no data language at all, so that users must explicitly describe data constructors and accessors. The tool should provide higher-level semantics in its primitive operations, and supply its own refinement to the actual primitives supported in code generation.

Both problems can be addressed by improving the user constraint language. One option would be to allow relatively general pseudocode, build a parser for it, and map traditional pseudocode concepts onto Designer concepts. Because of the desire for a graphical interface to Designer discussed in section 8.6, a better option would be to provide an iconic representation of the constructs traditionally provided by pseudocode. There are examples of this in existing commercial CASE tools [26]. It should be possible to construct a visual constraint editor that would communicate the traditional concepts and constructs to the user, but in a declarative paradigm. Iconic languages can also be assigned a direct semantic meaning by Designer in its own computational model.

In either case, it is clear that the computational constructs for choices and loops should be completely hidden from the user. Designer should automatically construct the nodes and structures it needs from some different, user-oriented representation.

### 8.6 CODE IMPROVEMENTS

Designer is written in VPI Prolog, in an object-oriented style, using a LISP-like syntax. It is inefficient in many ways. The Prolog code it generates has only
been examined for small examples, and is not yet trustworthy enough for production use. The entire system should be upgraded from a proof-of-concept to a reliable, production-quality software system. Although the code was developed using rapid prototype techniques, Designer was written to make it easy to upgrade and extend the program.

The HCI of Designer is very primitive. Its main feature is its reuse of the SITE HCI. An obvious improvement would be to provide a graphical graph editor.

The current prototype is required to operate on the customer’s ASCII text terminals. However, the current text-oriented syntax could be easily replaced by a different syntax. The existing SITE code was wrapped in an object-oriented user interface agent. Current HCI standards and techniques were applied to the design of this agent. It uses a declarative knowledge base that records the characteristics and abilities of the text-oriented HCI. All interaction with the user is handled by a set of object editors. These editors request generic services from the user interface agent, which maps the requests onto the available SITE services.

Constant values are created by using SITE data structures. This should be reworked, to represent constant values and the procedures necessary to derive them in a way consistent with the rest of Designer. Along with this enhancement, primitive data domains should be examined, to determine if Designer could benefit by representing them more simply than as first-class conceptual nodes.

Concepts are represented as frames. This permits many extensions to the knowledge associated with concepts. Some examples include Dasgupta’s history and rationale for design decisions, if-used and when-needed slots, and simulation classes.
Designer assumes that the provided built-in constraints are sufficient for the lowest level primitive computations. Experiments with design problems should determine if this assumption is true for real users.

The generic editor class is a good feature because it edits all objects in a consistent way. However, that same feature makes it difficult to recognize what kind of object is being edited. Better visual distinction among editors and data in the editors is needed.

8.7 IMPROVED CODE GENERATION

Generating code in languages other than Prolog is essential to practical acceptance of Designer. The and/or computational model is very simple; therefore, it should be straightforward to generate any modern, high-level language from a conceptual map. The initial support and interest in this thesis originate from the AEGIS project; therefore, the first languages added should probably be Ada and CMS-2.

This report is silent about the quality of the generated code. This issue should be addressed when Designer is used in professional applications. Other properties of generated code become more important if Prolog code is to be translated to code in other, procedural languages. For example, Prolog failure has no direct equivalent in Ada. A future task should examine what constraints Prolog code should satisfy in order to be easily translated to Ada. Another task should ensure that Designer generates Prolog code that satisfies these constraints.
8.8 SUPPORT FOR TRADITIONAL CASE PRODUCTS

At least some products of traditional CASE tools could be generated from a Designer conceptual map. A future project could provide a module to generate structure charts for use by other CASE tools. A standard interface, such as one that conforms to the Portable Common Tool Environment (PCTE) and CASE Data Interface Format (CDIF) standards, might be most useful. Some small experiments might indicate if Designer provides better information for simulators and analyzers than other CASE tools.

Existing CASE tools might also assist with the problem of deriving traditional software engineering documents from a Designer conceptual map. This is probably a necessary step to introducing Designer to existing software projects.

In traditional CASE tools, different views use very different features to describe a system (e.g., data flow, procedure calls). Transforming requirements to design, or generating code from a collection of views, requires metaknowledge about relationships between these different features. These features must exist, in some sense, within Designer’s conceptual map. Derivation of traditional CASE products may require their identification. This may also contribute to the problem of recognizing a semantic change in the design.

To accomplish this, the metaknowledge about design must be extended. In particular, Designer’s metaknowledge currently understands relatively low-level features (e.g., loops). Data flow and module cohesion represent a much higher level of abstraction.
8.9 EXTENDING DESIGN METAKNOWLEDGE

Designer uses metaknowledge about goals and constraints for conceptual maps, about mapping states of conceptual maps onto operations to bring those states closer to goal states, and about editing operations and their effects. This is all represented as heuristic knowledge, built into the tool.

N86 has expressed interest in making the design metaknowledge editable by end users. The metaknowledge fits easily into the same knowledge structures used for the target design knowledge, as demonstrated in Figure 10. The existing editors could be extended for this purpose. One topic for research, suggested by N86, is to express the design metaknowledge rules as ERA diagrams, to maintain them by a graphical editor.

It would be more difficult to update the editing metaknowledge. Adding knowledge about editing operations implies extending the abilities of the map editor itself, and is probably more suitable to system maintainers than to designers.

N86 has also expressed interest in using the design evaluator to test the legality of individual user edit actions before applying them. This coincides with the desire for the evaluator to run as a parallel agent. To support this new ability, it would need to cooperate closely with the editors. The editor and evaluator processes would have independent agendas, but need to exchange synchronizing messages regularly.

Section 8.1 discusses proposed applications that would make use of the Designer knowledge base to infer properties of a conceptual map beyond the immediate design constraints. N86 has enquired about the feasibility of doing these inferences as part of Designer, while the map is being edited. That approach would
also fit into a parallel agent implementation. It applies pressure to make the evaluator more of an independent application, while that the desire to use it for interactive editing checks applies pressure to make it more tightly coupled with the editors.

One solution to this tension is to make the evaluator a constraint server, capable of operating on multiple knowledge bases for multiple clients.

8.10 MACHINE LEARNING FROM DESIGN EXPERIENCE

The Computational Theory of Design uses Schön’s model concerning how humans approach problems. It only partially represents Schön’s insight that professionals use their experience to extend the theory of their profession. It would be very interesting to make Designer learn from the strategies applied by humans. Such an enhancement would make use of AI specialties not considered in this thesis. It would require a persistent, updatable knowledge base of design metaknowledge.

8.11 GRAPH THEORETICAL ANALYSIS OF CONCEPTUAL MAPS

The graph representation opens the possibility of applying techniques and principles of graph theory to design. One example is the problem of searching for duplicate meanings in the concept map. If Designer could recognize semantically congruent subgraphs, they could be checked for consistency or automatically merged.
The work described in this thesis has produced a theoretical description of generic design. The implementation of that theory as the design support tool Designer suggests that the theory is suitable for the domain of software development. Experiments with the prototype of Designer show that users can easily understand and work with conceptual maps, which form the base for the theory and implementation. The successful implementation of the prototype suggests that the goal of automated support for design at all levels of abstraction in a single knowledge structure is feasible. Examples were also generated of the practical generation of Prolog source code directly from the design structures. However, the Designer prototype fails to communicate with users about the constraint annotations of a conceptual graph.

Current specific plans for Designer include a cost/benefit study for N86 of alternative architectures and supporting hardware platforms. This will result in a recommendation regarding the options discussed in section 8.5 and 8.6. If N86 chooses to proceed with that recommendation, it is expected to be in support of the specific experiments and applications described in section 8.1. The other issues of this chapter remain topics for future discussion with N86.
CHAPTER 9
REFERENCES


23 Miller, G. The Magical Number seven, Plus or Minus two: Some Limits of our Capacity for Processing Information. Psychological Review, 63, 81-97.


References 213


APPENDIX A
EXPERIMENTAL PROTOCOL

The following standard protocol was given to each user in the evaluation experiments.

Design Tasks

1. From your design project, pick one small, but interesting subtask. It should satisfy a system requirement to perform a specific, well-understood computation.

2. Start the Designer program with the command "designer". At the Prolog ?- prompt, type "go".

3. From the conceptual map editor, add concept nodes to represent the concepts and data that are required to perform the target computation. To limit the experiment to an hour or less, you should use only 5 to 10 concepts.

4. Rename your conceptual map. This will make it possible for your map to be saved to a file using a name of your choosing.

5. Edit each concept in your design (node in the conceptual map). In the concept node editor, create links to specify which other concept nodes are used by the node you are editing. Exit the concept node editor when you finish each node.
Because Designer uses a text interface and shows links for only one node at a time, you may find it helpful to take a few minutes and sketch a diagram for the overall conceptual map.

6. The lowest-level concepts supply primitive data to the design, and therefore have no other concepts supporting them. Edit each of these nodes. In the concept node editor, edit a procedure for the node. In the procedure editor, construct a constant value for the concept.

7. Each of the other concepts in your design create a value by using their supporting concepts. Edit each of these concepts. Edit the node's procedure. In the procedure editor, construct a body for the procedure. A procedure body is a short block of pseudocode that specifies how to create a value for the concept, using the values of the lower-level concepts that support it. If appropriate, also construct a guard expression to control whether the body is executed all the time.

8. Return to the conceptual map editor. Use the autoRefineMap command to have Designer look for conflicts and missing detail. For a simple conceptual map, you may see no messages resulting from this step.

9. Save the conceptual map.

10. Generate code for your design.

11. Exit Designer.
APPENDIX B
EXPERIMENTAL DATA

B.1 DATA COLLECTION FORM

User's Reaction to the Designer tool

User's name:

For each question in the left-hand column, mark a score for Designer in the right-hand column.

<table>
<thead>
<tr>
<th>On-screen instructions and prompts</th>
<th>confusing</th>
<th>clear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connection between actions and results</th>
<th>confusing</th>
<th>clear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information displayed in a consistent manner</th>
<th>never</th>
<th>always</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence of displays</th>
<th>illogical</th>
<th>logical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5</td>
<td></td>
</tr>
</tbody>
</table>

Experimental Data 219
<table>
<thead>
<tr>
<th>Category</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error messages helpful</td>
<td>never</td>
<td>always</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finding data: navigating through a conceptual map</td>
<td>confusing</td>
<td>clear</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Changing information in a conceptual map</td>
<td>complex</td>
<td>simple</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response time</td>
<td>slow</td>
<td>quick</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usefulness of conceptual maps as a way to do design</td>
<td>poor</td>
<td>good</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ability of this tool to produce accurate designs quickly</td>
<td>poor</td>
<td>good</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effectiveness compared to other design tools</td>
<td>worse</td>
<td>better</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overall reaction</td>
<td>unfavorable</td>
<td>favorable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other comments:

Experimental Data
B.2 RESULTS

Designer Tool Experiments
User Reaction Survey

**Table 4:** Designer Prototype
User Qualitative Reaction Survey Results

<table>
<thead>
<tr>
<th>Item</th>
<th>User</th>
<th>mean</th>
<th>standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Response time</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Error message helpful</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Usefulness of maps</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Sequence of displays</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Finding data</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Information consistent</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Produce design quickly</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Changing information</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Connection actions/results</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Effectiveness/other tools</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Overall reaction</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Onscreen instructions</td>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
APPENDIX C
PROBLEM REPORTS

1. Designer sometimes does not display the currently-selected object in any editor.

2. The concept editor presents a confusing object display.

3. The relationship editor should present selections for support/refine and forward/backward as a menu. The user should pick a selection, not repeat the prompt.

4. The relationship editor does not always present the pick list of available concepts properly.

5. Different editors sometimes use the term "concept", sometimes "node". They should be consistent.

6. The editors do not display enough context to make it clear where the user is in the tool, and what relationship the currently edited object has to its owner or other related objects. All editors need more information and visual distinction.

7. The constraint editor requires the global context be entered by the user. It should be handled automatically.

8. The constraint editor should make clear what should be done there. Templates of constraint syntax and primitive predicate functor names would be helpful.
9. If the conceptual map editor receives the command editNode with a non-existent node name as argument, the concept editor displays the non-existent node. It should create it first, and only after user confirmation.

10. Several SITE editors present "exit" as a command choice. All these respond to exit by exiting the entire tool, without saving data. These must all be fixed.

11. The default conceptual map is clumsy. The conceptual map editor should either offer to load a map when none exists, or find a better display for an empty default map.

12. The constraint editor shows the text of the guard on first entry, but fails to display either guard or body after they have been edited. This editor needs more feedback about its data structures.

13. Class concept-procedure’s method asView generates a text representation that disagrees with the user entry syntax for the same class.

14. After exiting the constraint editor, the spurious message "failed to handle editNode." is displayed.

15. All editors should give better feedback about the results of their commands. Some commands (e.g. autoRefineMap, generateCode) are completely silent.

16. The output file for generated source code should be a unique name based on map name.
17. The code generator fails for some nodes. There is a bug in the validator for a subgoal, but it sometimes succeeds.

18. The evaluator does not return for some maps. There is an undiagnosed bug somewhere in it.

19. In the conceptual map editor, deleteNode of the only node in the map seems to hang the system. It uses no additional memory to do so. There is an undiagnosed bug somewhere in it.

20. After exiting the constraint editor, the spurious message "unknown class failed to handle command" appears if an incorrect command syntax was used before entering the constraint editor.

21. The conceptual map editor does not always direct edit actions generated by the edit generator to the appropriate editor for execution.
APPENDIX D
DESIGNER USER'S GUIDE

Designer Prototype
User's manual
February 21 1994

USAGE NOTICE:

Designer was built with resources of Computer Sciences Corporation (CSC) and the United States Navy. Copyright and use are governed by federal regulations regarding products created under contract to the U.S. Government. CSC retains all permitted rights. For information regarding access to Designer, contact

Robert Bartholow, Branch N86
Naval Surface Warfare Center
Dahlgren, Virginia 22448

Per agreement with NSWC, Designer and its source code have been delivered to Virginia Polytechnic Institute for academic use.

INTRODUCTION

Designer is very much a prototype. Its problem report book contains outstanding bugs. The Human-Computer Interface (HCI) is very primitive. The program works only well enough to show that the Computational Theory of Design is
sound, and that a practical design assistant tool can be built. Designer itself needs work to be practical.

This manual, therefore, is only a set of hints at the moment. Comments are welcomed (good and bad, please).

In this guide, Designer prompts are printed in this font. User responses are printed in this font.

For more information, consult the thesis "Supporting Design: A Computational Theory of Design and its Implementation in a Software Support Tool."

CONTENTS

1. STARTING UP ............................................................. 226
2. BUILDING A CONCEPTUAL MAP .................................... 228
3. DESIGN HINTS ............................................................ 236
4. PERFORMANCE ISSUES .................................................. 236

1. STARTING UP

System Requirements

Designer is written in VPI Prolog. It runs under VPI Prolog V1.041 and later. Earlier versions of VPI Prolog contain bugs that Designer hits.

Designer consumes excessive memory. Tell VPI Prolog to allocate sufficient heap space (with the prolog -h number startup flag). For a rough guide, use 300,000
bytes per node in the map. It cannot be run usefully on machines with less than 4 Megabytes.

**Running Designer**

To start designer, consult its Prolog source files. The VPI Prolog environment file seems to cause crashes. The file loader.he will do this:

`prolog loader`

At the VPI Prolog prompt, enter the query goal 'go':

`?- go`

This will initialize Designer, create a default initial conceptual map, and enter the conceptual map editor.

To quit Designer, from the conceptual map editor, enter the command 'exit'. This will return the VPI Prolog prompt. Enter the query goal 'quit' to exit Prolog:

`?- quit`

Designer User's Guide
2. BUILDING A CONCEPTUAL MAP

The HCI to Designer is a set of text editors. Designer works with a hierarchy of data structures:

c conceptual map
concept node
constraint on a concept (Designer calls it a procedure)
blocks of subgoals in a constraint (guard and body blocks)
text block

Each editor presents a menu of available commands. Always enter the complete command as it appears in the menu. All commands take optional parameters, to indicate the object to be manipulated. The menus give no hints about parameters, because Designer will prompt for them. In those cases where parameters make no sense for a command, Designer will use the command and ignore the parameters. For example:

selectNode  createLoop  createMap  createNode  deleteMap
deleteNode  loadMap    editNode    renameMap  saveMap
autoRefineMap  generateCode  exit

Command: createNode new-node-name

Designer User's Guide  228
Alternatively, giving the command alone causes Designer to prompt for the argument:

```
selectNode  createLoop  createMap  createNode  deleteMap
deleteNode  loadMap    editNode   renameMap  saveMap
autoRefineMap  generateCode  exit
```

Command: `createNode`

Name of new node: `new-node-name`

**Conceptual Map Editor**

This editor uses the screen:

```
Edit conceptual map
Nodes have links in two dimensions, seen when a node is edited.
Conceptual map: default-conceptual-map
Nodes:
top-level-construct
selectNode  createLoop  createMap  createNode  deleteMap
deleteNode  loadMap    editNode   renameMap  saveMap
autoRefineMap  generateCode  exit
```

It displays the list of concept nodes currently in the map. Working with any concept requires entering the concept editor with the command `editNode`.  

Designer User’s Guide 229
Notes on the commands:

selectNode [node-name] sets [node-name] as the current selected node. This is the focus of attention; the editor's cursor. Other commands will use the selected node as their default parameter.

createLoop assists the user to automatically set up a loop with an appropriate entry point, base clause and recursive clause. It currently doesn't give much practical help.

createMap creates a new, empty map.

createNode has a bug. Be sure to enter its parameter: the name of the new node to create. If the parameter is missing, it will crash the program rather than prompt for the node name.

euditNode runs the Concept editor.

autoRefineMap runs the Evaluator module. It applies a set of design constraints to the features (map, concepts, loops) of the map. When it finds constraint violations, it sends messages to the Edit Generator module, which looks up the violation in a knowledge base and returns a suggested edit to correct the violation. Warning: autoRefineMap may never return for a large map.

generateCode sometimes skips some nodes. It currently always writes to the same file name, "map.pro". It writes standard Edinburgh syntax.
Concept Editor

This editor uses the screen:

Edit concept

Concept: top-level-construct
Description:
nil
Supported by:
a
b
Supports:
c
Refined by:
t1
t2
Refines:
Procedure is good
createNode createLink deleteLink deleteNode
editDescription editProcedure moveLink exit

It displays all of the nodes linked to the current node, grouping them by the four possible links: supports another node, supported by another node, is a refinement of a parent node, is refined to child nodes. The comment "Procedure is good" indicates that the concept's annotated procedure has passed its local validity constraints.

Designer User's Guide
Notes on the commands:

createLink [link-type] [linked node] is the only way to create a new link in the map. Designer supports two types of links, support links and refinement links. See the thesis for more on the difference between them. Briefly, refinement supports hierarchical breakdown of a map. For small experiments, only worry about the support links. This command will prompt for everything it needs. Type one of the choices displayed, exactly as it is shown.

moveLink [old node] [new node] moves an arc on the map. This would be much easier to understand in a graphical editor. The node currently being edited has a link to something else, e.g.

current \rightarrow a

moveLink picks up the other end of that arc and move it to a different node, e.g.

\texttt{moveLink a b}

results in

current \rightarrow b

editProcedure runs the Procedure Editor.
Procedure Editor

This editor uses the screen:

**Edit concept-procedure**

A procedure body is executed only if its guard is true.

Procedure for concept: top-level-construct
returns type untyped
variables

Guard:
    guard := true( globalIn globalOut)
Body of Procedure:

constructLoop   constructOr constructBody   constructGuard
constructConstant seeBuiltIns editDescription declareType
exit

Notes on the commands:

constructLoop see the notes on this command in the Conceptual Map editor.

constructOr Choice (if/then/else) is encoded in the map, viewing it as an
and/or graph. This command declares that the current node is an
or-node, so that its procedure uses the value of exactly one of its
children. By default, a node is assumed to be an and-node, so that its
procedure contains a computation that uses all of its children.

constructBody/constructGuard each of these runs the Procedure Constraint
Editor. The guard and body of a procedure are each a constraint
expression, made up of a list of clauses. See the thesis for more.
constructConstant uses SITE's knowledge acquisition code to help the user build a primitive value for a concept. This command will let the user construct either:

- a literal data structure as the constant value of the procedure. In this case, the concept is a leaf node.

- a query to the end user to enter a constant value, which is then used as the value of the procedure. In this case, the concept is a leaf node.

- a query to the end user to select one from a list of constants. The list must be the constant value of a supporting node. The supporting node is a leaf node, whose constant value was already built using this command.

seeBuiltIns shows the list of primitive constraints supported by Designer. Currently not very extensive.

Procedure Subgoal Block Editor

Unlike the other complex class editors, this editor is a simple text editor. It accepts any input, and the input is parsed to check for correctness after the editor is exited.
The syntax to express a constraint is intended to be familiar to users. Briefly, it interprets a constraint as a list of assignments. This turned out to be a bad idea, and will change in the future. The required syntax is

\[
\begin{align*}
\text{constraint} & : = \text{goals} \\
\text{goals} & : = \text{goal goals} \\
& : = \langle \text{empty} \rangle \\
\text{goal} & : = \text{variable-name} \ ':=' \ \text{functor} \ '( \ \text{args} \ ')' \\
\text{variable-name} & : = \langle \text{any atom} \rangle \\
\text{functor} & : = \langle \text{name of a supporting node} \rangle \\
& : = \langle \text{name of a primitive constraint supplied by Designer} \rangle \\
\text{args} & : = \text{user-args} \ \text{globalIn}'\',\ \text{globalOut} \\
\text{user-args} & : = \text{user-arg}'\',\ \\
& : = \langle \text{empty} \rangle \\
\text{user-arg} & : = \langle \text{variable that is already assigned in this constraint} \rangle \\
& : = \langle \text{input variable supplied by the parent node's procedure} \rangle
\end{align*}
\]

Each clause is a predicate/function on values that are already known. Every concept has a single value, so every clause that executes a supporting concept results in a single value being returned, which is bound to the variable on the left-hand side of the assignment.

The arguments to each clause represent values that have already been computed. The two special values globalIn and globalOut represent variables that are passed around throughout the computation. They are necessary to achieve Turing-completeness, and give access to global context. It was not necessary to force
the user to type them in every clause. This will be handled automatically in the future.

3. DESIGN HINTS

Designer is written in an object-oriented style (not dialect) of Prolog. Its design architecture is found in Chapter 5 of the thesis.

Many of the Prolog predicates are generic, polymorphic methods. This generally means that they construct metapredicates to determine the appropriate predicate. The directory with the Designer source code includes the design file object.dsn. This should give some help. Look in the source file object.hc for where most of the real work happens. If something appears to disappear off the edge of a clause, it is likely inheriting a behavior from its superclass. When the clause fails, the actual last clause of the same functor calls the clause

(object nomethod ?Class ?Args)

which attempts to determine the superclass and pass the message up to it.

4. PERFORMANCE ISSUES

Designer is slow. It runs barely fast enough on a large Sun, almost too slow on a VAX 6460, and too slow on a 68000.
Vita

Glenn Holliday was born in North Carolina on 1 December 1955. He was graduated from Virginia Polytechnic Institute with a Bachelor of Arts in Communications in 1977. He has worked as a professional software engineer since 1982. He entered the graduate program in Computer Science at Virginia Polytechnic Institute in 1986. Since 1991, he has been Lead Programmer of the Knowledge Engineering group at Computer Sciences Corporation’s King George, Virginia office. His interests are intelligent derivation of design information, logic programming, and agent architectures.

Glen Holliday