INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark □.

1. Glossy photographs or pages □
2. Colored illustrations, paper or print □
3. Photographs with dark background □
4. Illustrations are poor copy □
5. Pages with black marks, not original copy □
6. Print shows through as there is text on both sides of page □
7. Indistinct, broken or small print on several pages □
8. Print exceeds margin requirements □
9. Tightly bound copy with print lost in spine □
10. Computer printout pages with indistinct print □
11. Page(s) __________ lacking when material received, and not available from school or author.
12. Page(s) __________ seem to be missing in numbering only as text follows.
13. Two pages numbered __________. Text follows.
14. Curling and wrinkled pages □
15. Other ____________________________________________________

University
Microfilms
International
THE IDENTIFICATION OF SEMANTICS FOR THE FILE/DATABASE PROBLEM
DOMAIN AND THEIR USE IN A TEMPLATE-BASED SOFTWARE ENVIRONMENT

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

by
Charles John Shubra Jr., B.S., M.S.

* * * * *

The Ohio State University
1984

Reading Committee:

Dr. Jayashree Ramanathan
Dr. Stuart H. Zweben
Dr. Bruce W. Weide

Approved By:

Jayashree Ramanathan
Adviser
Department of Computer and Information Science

1 This research was supported in part by the NBS grant AQ4580.
Dedicated to

Sue, Beth and Chris
My Mother, and especially
My Father, whose memory gave me the will to continue
ACKNOWLEDGMENTS

I wish to thank Jay Ramanathan, my advisor, for her professional guidance, boundless energy and unending encouragement during my graduate career. Further, and more importantly, I wish to recognize her and her family for the many kindnesses extended to me, not the least of which was their hospitality. They opened their home and hearts in a manner that will always be a model for me. I would like to thank Bill Hochstettler and Zeke Haley who offered shelter and friendship during my visits to Ohio State, particularly during that long hot summer of '82.

I would also like to thank the members of my committee, Stu Zweben and Bruce Weide for their careful reading and helpful suggestions.

Three corporations played a major role in this research and for their support I wish to thank them. Particularly, I wish to thank Dr. R. Fidoten and Ms. S. Galli of PPG Inc., Mr. A. Czapic of Mellon Bank Inc., and Mr. L. Domensic of Westinghouse Telecomputer Center.

I wish to recognize the members of the TRIAD Research Group past and present, particularly Lucy Wright, Chin Li, Dilip Soni, Jeremy Kuo, Jim Kiper, Walt McKnight, and Steve Demion, for sitting through endless presentations and offering valuable suggestions. I
extend thanks to Marty Marlatt and Carol Miller for helping in preparation of this document.

Finally, I wish to thank and extend my love to my family Sue, Beth, Chris and Binki who encouraged me in my efforts and understood when daddy had to be in Ohio working.

Thank all of you. Your kindnesses will be with me forever.
VITA

1969 ....... B.S., Mathematics, Indiana University of Pennsylvania, Indiana, PA
1970 ....... M.S., Computer Science, The Pennsylvania State University, University Park, PA
1973-1974 ... Programmer Analyst, Corning Glass Works, Raleigh, NC
1974-1978 ... Associate Professor, Computer Science
1980-present .. Department, Indiana University of Pennsylvania, Indiana, PA

PUBLICATIONS


RESEARCH INTERESTS

Software Engineering: software development methodologies, software environments, patterns of programming
Database Management: data model design, DBMS software packages, end-user languages

Management Information Systems: analysis and design techniques, end-user interfaces, decision support systems
TABLE OF CONTENTS

1. Introduction 1
   1.1 Motivation 1
   1.2 Template Approach to Patterns of Programming 7
   1.3 Survey of Related Research 15
      1.3.1 Conceptual Models Applicable to the Domain of Study
         1.3.1.1 Jackson and Warnier Methodologies 16
         1.3.1.2 Algorithms + Data Structures = Programs 18
         1.3.1.3 Algorithms = Logic + Control 18
      1.3.2 Human Factors Experimentation 20
      1.3.3 Reusable Software Components 22
         1.3.3.1 Identification of Reusable Components 23
         1.3.3.2 Role of Components in Software Psychology 26
         1.3.3.3 Existing Efforts to Utilize Components in Program Development 27
      1.3.4 Artificial Intelligence Approach to Program Environments 33
      1.3.5 Database Management Systems Software and Descriptive Languages
         1.3.5.1 Data Manipulation Language Facilities of DBMS 37
         1.3.5.2 Query Language Facilities and Fourth Generation Languages 38
      1.3.6 Grammar Theory and Templates 43
   1.4 Scope of the Thesis and Contributions 47

2. Formation of a Conceptual Model of the Problem Domain 51
   2.1 Introduction 51
   2.2 Preliminary Study of Fourteen Student Programs 52
   2.3 Controlled Experiments Based on Findings from the Preliminary Study
      2.3.1 Experiment 1 -- Variety in Control Structures 61
      2.3.2 Experiment 2 -- Effects of Pattern Familiarity on Performance of Maintenance Tasks 65
   2.4 Conclusions Drawn from the Two Experiments 78

3. Case Study to Discover File/Database Problem Domain Semantics 80
3.1 Case Study
3.1.1 General Data 81
3.1.2 Rationale for the Case Study 84
3.1.3 Techniques Used in the Case Study 84
3.1.4 Hypothesis Template Set 85
3.1.4.1 Site and Program Selection 89
3.1.4.2 Program Analysis Techniques Used 90
3.2 Semantic Characterization of the File/Database Domain 92
3.2.1 Concepts Used in the Characterization of the Domain 93
3.2.2 Impact of the Case Study 95
3.2.2.1 Addition of Semantics to Individual Templates 95
3.2.2.2 Completeness of the Enhanced Template Set 96
4. Description and Discussion of Semantics for a Representative Template
4.1 Summary of the Validate - V Template 102
4.1.1 Related Templates 102
4.1.2 Terminology 103
4.2 Discussion of Validate Semantics 106
4.2.1 Template Semantics and Related Case Study Observations 107
4.2.1.1 Record Sources and Destinations 107
4.2.1.2 Tests 108
4.2.1.3 Reporting 117
4.2.1.4 Disposition of Records After Completion of Testing 119
4.3 Overview of Programs Using the Validate Template 121
4.4 Summary Data Collected for Programs Containing the Validate Template 123
4.5 Comparison of Semantics 132
5. Organizing and Representing Template Knowledge 140
5.1 Organization of the Semantics in the File/Database Processing Domain 141
5.1.1 The Relationships Among Templates 141
5.2 Semantics of Individual Templates 152
5.3 Representing Knowledge in Templates 155
5.3.1 Attributed Grammar Form Model 155
5.3.1.1 Traditional Use and Evaluation of Semantic Functions 156
5.4 Representation of the Knowledge in the Generalized Traversal and Validate Templates 160
5.4.1 Definition of a Preliminary Version of the I Template Grammar 160
5.4.2 Enhancing the I-Structural Grammar 163
5.4.3 The Process of Creating a Template Representation  
5.5 Attribute Grammar for the Validate – V Template  
5.6 The Use of Artificial Intelligence Techniques  

6. Application of Several Techniques to a Problem for the Purpose of Comparison  
6.1 Specifications for the Problem to be Solved Using Each Technique  
6.1.1 Problem Requirements in Terms of Validate Semantics  
6.2 Logrippo Technique Applied to the Problem  
6.2.1 Skeletal Code Generated by the Logrippo Technique  
6.2.2 Discussion of Logrippo Results  
6.3 Jackson Methodology Applied to the Problem  
6.3.1 Skeletal Code Suggested by the Jackson Methodology  
6.3.2 Discussion of Jackson Results  
6.4 Generalized Traversal – I Template Applied to the Problem  
6.4.1 Skeletal Code Generated by the I Template  
6.4.2 Discussion of I Template Results  
6.5 Validate – V Template Applied to the Problem  
6.5.1 Skeletal Code Generated by the Validate Template  
6.5.2 Discussion of Validate Template Results  

7. Use of the Knowledge in Templates  
7.1 Overview of the System Architecture  
7.2 Scenario: Interaction with the Template-Set Presentation Expert  
7.2.1 Scenario Part A  
7.2.2 Scenario Part B  
7.2.3 Scenario Part C  
7.2.4 Scenario Part D  
7.2.5 Scenario Part E  
7.3 Issues Related to the Use of Templates  
7.3.1 Template Combinations and Interconnections  
7.3.2 I/O Optimization  

8. Conclusions  
8.1 Future Research Directions  

Appendix A. Human Factors Experiment 1 Materials  
Appendix B. Materials Used Human Factors Experiment Two  
Appendix C. Template Descriptions and Summary Data Resulting from the Case Study of Production Programs  
C.1 Generalized Traversal Template – I
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1.1 Template Semantics and Related Case Study</td>
<td>284</td>
</tr>
<tr>
<td>Observations</td>
<td></td>
</tr>
<tr>
<td>C.1.1.1 Traversal of the Hierarchy of Objects</td>
<td>284</td>
</tr>
<tr>
<td>C.1.1.2 The Criteria for Grouping Key-Related Records</td>
<td>291</td>
</tr>
<tr>
<td>C.1.1.3 The Effect of Multiple Record Types on File Traversal</td>
<td>292</td>
</tr>
<tr>
<td>C.1.1.4 Generation of Access Statements for the Nested Hierarchy</td>
<td>294</td>
</tr>
<tr>
<td>C.1.2 Overview of Programs Using the Generalized Traversal Template</td>
<td>303</td>
</tr>
<tr>
<td>C.1.3 Summary Data for Programs Using the Generalized Traversal Template</td>
<td>304</td>
</tr>
<tr>
<td>C.2 The Fabricate Template - Y</td>
<td>305</td>
</tr>
<tr>
<td>C.2.1 Template Semantics and Related Case Study</td>
<td>306</td>
</tr>
<tr>
<td>Observations</td>
<td></td>
</tr>
<tr>
<td>C.2.1.1 Input Data</td>
<td>306</td>
</tr>
<tr>
<td>C.2.1.2 Output Data</td>
<td>307</td>
</tr>
<tr>
<td>C.2.1.3 Fabrication Actions</td>
<td>307</td>
</tr>
<tr>
<td>C.2.2 Overview of Programs Using the Fabricate Template</td>
<td>310</td>
</tr>
<tr>
<td>C.2.3 Summary Data for Programs Using the Fabricate Template</td>
<td>311</td>
</tr>
<tr>
<td>C.3 Report Generation Template - X</td>
<td>315</td>
</tr>
<tr>
<td>C.3.1 Template Semantics and Related Case Study</td>
<td>316</td>
</tr>
<tr>
<td>Observations</td>
<td></td>
</tr>
<tr>
<td>C.3.1.1 Major Components</td>
<td>316</td>
</tr>
<tr>
<td>C.3.1.2 Source(s) of Data</td>
<td>316</td>
</tr>
<tr>
<td>C.3.1.3 Report Categories</td>
<td>319</td>
</tr>
<tr>
<td>C.3.1.4 Report Formats</td>
<td>321</td>
</tr>
<tr>
<td>C.3.2 Overview of Programs Using the Report Template</td>
<td>326</td>
</tr>
<tr>
<td>C.3.3 Summary Data for Programs Using the Report Template</td>
<td>328</td>
</tr>
<tr>
<td>C.4 Templates Which Initialize Tables - A, B, J</td>
<td>331</td>
</tr>
<tr>
<td>C.4.1 Template Semantics and Related Case Study</td>
<td>333</td>
</tr>
<tr>
<td>Observations</td>
<td></td>
</tr>
<tr>
<td>C.4.1.1 Structure of Cobol Tables</td>
<td>334</td>
</tr>
<tr>
<td>C.4.1.2 Structure and Content of the Source of Data</td>
<td>335</td>
</tr>
<tr>
<td>to be Placed into the Table</td>
<td></td>
</tr>
<tr>
<td>C.4.1.3 Method of Placing Initialization Data into the Table</td>
<td>335</td>
</tr>
<tr>
<td>C.4.1.4 Table Reuse and Reinitialization</td>
<td>339</td>
</tr>
<tr>
<td>C.5.1 Template Semantics and Related Case Study</td>
<td>340</td>
</tr>
<tr>
<td>Observations</td>
<td></td>
</tr>
<tr>
<td>C.5.1.1 Placing Data into the Table</td>
<td>341</td>
</tr>
<tr>
<td>C.5.1.2 Obtaining Data from the Table</td>
<td>342</td>
</tr>
<tr>
<td>C.5.1.3 Summary Overview of Programs That Use Table Templates</td>
<td>343</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Count of Student Programs by Control Code Category</td>
<td>57</td>
</tr>
<tr>
<td>Table 2</td>
<td>Implementation of Process Functions in Student Programs</td>
<td>58</td>
</tr>
<tr>
<td>Table 3</td>
<td>Observed Frequency of Algorithm Selection in Experiment 1</td>
<td>64</td>
</tr>
<tr>
<td>Table 4</td>
<td>Observed and Expected Frequencies of Algorithm Selection</td>
<td>66</td>
</tr>
<tr>
<td>Table 5</td>
<td>Summary Data for the Familiar First Group</td>
<td>73</td>
</tr>
<tr>
<td>Table 6</td>
<td>Summary Data for the Unfamiliar First Group</td>
<td>74</td>
</tr>
<tr>
<td>Table 7</td>
<td>Estimates of Programmer Development Time Represented by the Case Study Sample</td>
<td>83</td>
</tr>
<tr>
<td>Table 8</td>
<td>Comparison of The Three Template Sets</td>
<td>87</td>
</tr>
<tr>
<td>Table 9</td>
<td>Templates Contained in the Enhanced Template Set</td>
<td>97</td>
</tr>
<tr>
<td>Table 10</td>
<td>Template Usage in the Program Sample</td>
<td>99</td>
</tr>
<tr>
<td>Table 11</td>
<td>Categories of Validity Tests</td>
<td>111</td>
</tr>
<tr>
<td>Table 12</td>
<td>Reporting Categories</td>
<td>118</td>
</tr>
<tr>
<td>Table 13</td>
<td>Disposition Actions</td>
<td>119</td>
</tr>
<tr>
<td>Table 14</td>
<td>Legend for Interpreting Notation Used in Reporting Summary Data</td>
<td>124</td>
</tr>
<tr>
<td>Table 15</td>
<td>Uses for Local-Plans in the Template Approach</td>
<td>158</td>
</tr>
<tr>
<td>Table 16</td>
<td>Productions in the I-Structural Grammar</td>
<td>162</td>
</tr>
<tr>
<td>Table 17</td>
<td>Relationships Among I Template and Validate Template Productions</td>
<td>173</td>
</tr>
<tr>
<td>Table 18</td>
<td>Limitations on the Interconnection of Templates</td>
<td>227</td>
</tr>
<tr>
<td>Table 19</td>
<td>Partitions Present in Figure 21</td>
<td>286</td>
</tr>
<tr>
<td>Table 20</td>
<td>State-Transaction Matrix Defining Legal Transaction Sequences for a Single Master Key</td>
<td>355</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A Hypothetical System/User Interaction Based on portions of the <strong>Validate Template</strong></td>
<td>13</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Sample <strong>Generic</strong> Code and <strong>Skeletal Source</strong> Code for the <strong>Validate Template</strong>.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3</td>
<td>&quot;Nested&quot; Condition Approach to Branch Identification</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4</td>
<td>&quot;Not Nested&quot; - Linear Approach to Branch Identification</td>
<td>55</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Local Loop in the Match Branch</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Refinements of Template Sets</td>
<td>86</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Hierarchy of Templates</td>
<td>142</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Generic Code Generated by the <strong>B Template</strong> Showing the Original Source of the Semantics</td>
<td>147</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Generic Code Generated by the <strong>Table Traversal-H Template</strong></td>
<td>148</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Generic Code Generated by the <strong>Table Search-G Template</strong></td>
<td>149</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Two Types of Mappings Used to Define a New Template</td>
<td>151</td>
</tr>
<tr>
<td>Figure 12</td>
<td>File Environment Diagram for the Problem</td>
<td>189</td>
</tr>
<tr>
<td>Figure 13</td>
<td><strong>SUBJECT-RECORD-SOURCE</strong> Data Structure Diagram</td>
<td>193</td>
</tr>
<tr>
<td>Figure 14</td>
<td><strong>ERROR-REPORT</strong> Data Structure Diagram</td>
<td>194</td>
</tr>
<tr>
<td>Figure 15</td>
<td><strong>VALID-RECORD-DISPOSITION-FILE</strong> Data Structure Diagram</td>
<td>195</td>
</tr>
<tr>
<td>Figure 16</td>
<td><strong>INVALID-RECORD-DISPOSITION-FILE</strong> Data Structure Diagram</td>
<td>195</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Program Structure Diagram for the Sample Problem</td>
<td>196</td>
</tr>
<tr>
<td>Figure 18</td>
<td>System Architecture</td>
<td>213</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Nested Source Lines From Two Templates</td>
<td>228</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Linear Sequence of Two Templates</td>
<td>229</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Three Sort Key - Three Group Hierarchy</td>
<td>285</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Three Sort Key - Four Group Hierarchy</td>
<td>285</td>
</tr>
<tr>
<td>Figure 23</td>
<td>A 5-Level Hierarchy</td>
<td>288</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Application Hierarchy with a Missing Intermediate-Level</td>
<td>288</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Codasyl DBMS Implementation of a Nested Hierarchy</td>
<td>289</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Two Structures Representing Multiple Record</td>
<td>293</td>
</tr>
</tbody>
</table>
Types on a Single Level of a Hierarchy

**Figure 27:** Traversal of the Structure Represented in Figure 26 Part 1

**Figure 28:** Alternate Traversal of the Structure Represented in Figure 26, Part 1

**Figure 29:** Priming-Read Algorithm

**Figure 30:** Traversal Where Delimiters are Used to Group Records

**Figure 31:** Jackson Diagram for a Structure That Uses Delimiters

**Figure 32:** Set BC entry Through its **Owner** B

**Figure 33:** Set CB Entry Through Its **Member** B

**Figure 34:** Set BC Traversal Using Owner Entry

**Figure 35:** Set CB Traversal Using Member Entry
Chapter 1

Introduction

1.1 Motivation

Software development and maintenance problems commonly referred to as the "software crisis" have been well documented [Boeh73], [Boeh76], [Lien78], [Wass78]. Many manifestations of the "crisis" have been reported with attendant speculation on the underlying causes and possible solutions [Floy79], [Zelk78], [Wass82], [Wino79]. The inclination of software developers to produce unique, creative solutions to recognizable, recurring application requirements has a detrimental effect on the productivity of both the developer and the maintenance programmer. For the developer the formation of a new solution is more expensive than reusing or adapting an existing proven solution. For the maintenance programmer discovery and understanding of unique solutions is time-consuming and error-prone.

The shortcomings of existing methods of software development and the failure to effectively build on past experience point to the need for applying engineering techniques to software production. In order to utilize an engineering approach, a foundation of knowledge
and techniques must be developed and organized. The formation of this body of knowledge and techniques, in other engineering fields, has evolved over years of research based on the discovery of controlling physical laws and a wealth of practical experience. Because software is a logical, not a physical entity, and because of the newness of the field, a definitive set of laws governing the process have not been formed. This new knowledge and these new techniques must be shown to be beneficial in constructing practical solutions to realistic applications.

The research in this thesis addresses the discovery and organization of knowledge based on existing software experience. The research process is enhanced if the scope of study is limited to a recognizable class of applications which share common characteristics. In important, yet narrow problem categories such as compiler construction and operating systems, techniques have been identified, taught, and in some instances automated to have a major positive impact on software development within those very special categories.

Within the universe of software, recognizable groupings of classes of applications, herein called problem domains, have been identified.

Definition: Problem Domain - a recognizable grouping of classes of applications that share common characteristics such as sources of complexity, problem requirements, solution techniques, etc.
It is meaningful to approach software research through such a partitioning because:

- Researchers generally agree that problem domains exist and they discuss software in terms of these problem domains. Some examples of problem domains are: file/data base processing (data structure oriented applications), scientific problem solving (computation oriented applications), real-time operations and process controlling (controlling the sequence and timing of process execution).

- Programming languages contain syntactic features tailored to the characteristics of problem domains. This includes Cobol's file handling constructs, and the concurrent features of Modula [Wirt77] and Concurrent Pascal [Hans75].

- Software development methodologies are more effective in some domains than others. For example, Jackson [Jack75] or Warnier [Warn74] methodologies are useful for file processing, where the R-Net methodology [Alfo76] is useful for the real-time processing domain.

- Computer Science curricula are organized, to some extent, around problem domain techniques and considerations [Adam81] and [Aust79].

- Artificial intelligence and expert systems research use problem domains as a means of focusing efforts in the construction of knowledge bases and inference rules. Examples of domains and implemented systems in this area are: medical diagnosis domain and MDX [Chan83b], Internist [Popl77], KMS [Regg80], Mycin [Davi77] systems; the oil exploration domain and the Dipmeter Advisor System [Davi81].

The existence of problem domains alone is not sufficient justification for their use in limiting the scope of this research. The need for manual analysis of a large amount of production software in order to identify problem classes within a problem domain, the semantics of those classes and the patterns of
programming cannot be met to any depth without limiting the scope of software to be studied.

Definition: Semantics - the concepts and characteristics embodied in a problem domain which are essential to the understanding of the domain and the design of solutions to problem requirements from the domain.

Research techniques appropriate for studying one domain may not be appropriate for another because the conceptual model used to develop the techniques may be domain-dependent.

Definition: Conceptual Model - A representation of the fundamental concepts and relationships among those concepts which are critical for understanding a problem domain. The conceptual model aids in the identification and categorization of knowledge about the problem domain by identifying the important concepts.

Further, not all problem domains have matured to the point where a sufficient body of experience exists.

Since the software crisis was a motivating factor for this research, a problem domain where the effects of the crisis are apparent was selected so that an engineering approach could be beneficial in solving problems. Such a problem domain is the file/database processing domain which has wide ranging impact in government, business and industry, and education. It has matured to the point where numerous methodologies have been defined for use in the domain, including Jackson [Jack75], Warnier [Warn74], Structured Design [Your79], PSL/PSA [Teic77], and SADT [Ross80]. These methodologies serve as a starting point for the construction of a
conceptual model of the problem domain. A large body of production software exists from which study samples can be obtained. Most, if not all, of the problems contributing to the software crisis are visible within this problem domain; therefore, the potential to benefit from an engineering approach is present.

**CHARACTERISTICS OF THE FILE/DATABASE PROBLEM DOMAIN**

The file/database processing domain which was selected to be studied in this research can be characterized by large volumes of records organized into several separate files and/or databases which need to be processed. Some of the processes deal with individual records or fields of a record but most deal with aggregates of records. Such aggregates are defined by relationships among records. In some cases, these relationships are represented in the stored structures and only need to be traversed. In other cases, these relationships are not represented in the stored structures, but require complex algorithms to form the needed data aggregates by traversing the stored structures. While database management systems such as IMS [IBM] and IDS [Hone80] have facilitated the formation of data aggregates and storage structure traversal, algorithms within this domain remain complex and difficult to construct.

Reusable program components or standard solutions to specific recurring application requirements -- patterns of programming -- are the basis for organizing knowledge about domains. Existing reusable
components lack the **ability to deal with common variations in requirements** and cannot **easily be combined** to form solutions to more complex requirements.

**Definition: Template** - a generalized syntactic unit which incorporates powerful semantics to support the programmer, both during program development and during maintenance.

Templates overcome the combination problems and inflexibility of reusable components while organizing the set of semantics pertaining to a class of problems within the file/database processing domain. Encoding of a set of templates using attributed grammar forms is shown to be a powerful yet user-friendly approach which facilitates automated support of software development and maintenance. This same set of templates could contribute to a knowledge-based system for constructing solutions.

Section 1.2 contains an overview of template terminology and concepts. The understanding gained in that section will make the survey of related research contained in Section 1.3 more meaningful and permits comparisons to be made with the research presented in this dissertation. Section 1.4 then details the scope of this thesis including contributions.
1.2 Template Approach to Patterns of Programming

The goal of the research in this thesis is to capture experience and knowledge of the file/database problem domain in a form that permits systematic, consistent use of the information in future program development. One means of gaining an understanding of problem domain knowledge is to identify and categorize patterns which recur in programs and problem specifications. The most available source of programming experience to study is program source code listings developed to solve problems in the domain.

Recognizing commonality in programs is a difficult task because at the source code level no two programs are alike. If recurring patterns are to be identified, it is necessary to reach a level of abstraction that overcomes superficial implementation language differences. Abstracting implementation differences results in the recognition of some commonality. In order to identify patterns in the form of commonly occurring problem requirements, further abstraction is needed. This additional abstraction overcomes the differences in the many possible designs that could be used to solve a given set of problem specifications.

As a result of research for this thesis, a series of conceptual models were developed, each more descriptive of the domain than its predecessors. The conceptual models were the basis of research techniques used in analyzing programs for patterns because they identified components critical to analysis. The conceptual model
also served as a framework for categorizing and organizing findings. The conceptual model which eventually evolved indicated that the following components were important in the domain:

- The data structures present,
- The correspondence between the data structures and the control code of the algorithm,
- The content and location of process code, and
- The semantics which added a depth of understanding to these previous components by dealing with specific problem class characteristics.

Armed with a descriptive conceptual model the problem domain was divided into problem classes based on existing attempts at reusable components or model programs. These existing attempts were found to contain superficial semantics. An extensive case study was undertaken to identify and classify the semantics in the problem domain. The semantics thus discovered were loosely organized by problem class.

Templates as defined here, were the vehicle used to impose a more rigid organization upon problem class semantics. Templates would ultimately be encoded into a form that could be processed on a computerized system thus becoming the foundation for an automated software environment. In the encoding of a template, a context-free

---

1 Recall that semantics refer to the concepts and characteristics of problem classes
grammar was used for those semantics which could be represented syntactically. Context-sensitive semantics were represented in a template by attributes and functions for manipulating attribute values. These functions were an extension of the semantic functions normally associated with attributed grammars. Since the functions defined for templates extend the traditional role of semantic functions, they will be called "plans."

One use of plans is to generate a skeletal program in a high-level programming language.

Definition: Skeletal Program - a partially completed program generated as the result of system/user interaction during which the user, responding to system generated prompts, provided application specific information. Skeletal programs, once completed by the applications programmer, become the program solution to specific application requirements.

In order to generate skeletal code the semantics of a problem class, are combined with the programming language in such a way as to allow automatic generation of a partially completed program (skeletal program). Formation of the skeletal program is directed by user-supplied, application-specific information. This process of the user supplying details in response to calls for information from the system is called "instantiation." During this process the template-based software environment constructs a derivation tree.

Definition: Generic Code - an attempt to illustrate the code generation aspects present in a specific derivation tree produced to meet a single set of application
requirements. Generic code is devoid of programmer supplied source code.

This derivation tree which is constructed and manipulated by the template plans is the source of program development data.

Because of the importance of templates to this research and the reader's understanding it, a motivating example of a template and the template concept is now given.

The definition of a template given in this section encompasses all the potential functions that a template should provide. A template can provide numerous functions as indicated by a sample Validate template in Figures 1, 2. These two figures are related and should be perused in parallel.
A template can:

**standardize the thought process by**

- guiding the programmer through a logical top-down organization of the problem and its solution (see Figure 1, item 1),

- presenting variations in specifications that must be considered at any point (see Figure 1 item 2),

- presenting checklists which comprise the generic solution (see Figure 1 item 3),

- collecting and organizing the documentation (see Figure 1 item 3),

**standardize the coding (recording) process by**

- providing a functional specification of a section of code (demonstrated by parallel line numbering in Figure 2),

- enforcing a standardized control structure for a class of problems (see Figure 2),

- providing a skeletal program to be completed by the programmer (see Figure 2).
Figure 2 shows how user responses can be padded by an environment designed to support the template, into skeletal code for an application. In Figure 2, the text in angle brackets indicates problem domain and template-specific prompts which can be used in documentation and in the organization of the actual code.

A template, when completed by the programmer, results in a derivation tree (part of which is shown as generic code) and a skeletal program (see example in Figure 2). Several templates might be used to complete a software project and occasionally, new templates might be required.

As indicated by Figure 2, a template-based approach could also facilitate better automated support. For example, a maintenance programmer could use a single command to display, instantaneously, the code for `<invalid_record_reporting>` in a template-based environment. This organizer, `<invalid_record_reporting>`, could be used to retrieve the code as indicated by the arrow in the Figure 2. A conservative estimate of the time taken to completely determine the semantics of the average program from the case study sample is 8 hours. This suggests that considerable time would have to be spent by a program maintainer to manually locate and retrieve the code which does invalid record reporting.
1. The Validate template contains the following major components:

- A source of records of questionable quality,
- A sequence of tests to determine the correctness of individual records and groups of records,
- A means of recording and reporting the test results,
- A flexible mechanism for disposition of tested records determined by validation test results.

2. Disposition of tested records can:

(Each of the following represent possible variations in the specification for a single task - record disposition)

- be specified as part of the individual validation test if record disposition does not depend on the validity of a group of records,
- be specified as part of a validity test for a group of records,
- be controlled by a separate disposition test, thus uncoupling disposition and any validity test.

3. For this record validity test you have:

<table>
<thead>
<tr>
<th>Checklist</th>
<th>User's Choice from many possible choices for each item in the checklist</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger test type</td>
<td>&lt;record type&gt;</td>
</tr>
<tr>
<td>validation test type</td>
<td>&lt;field completeness&gt;</td>
</tr>
<tr>
<td>reporting action</td>
<td>&lt;print error message when invalid&gt;</td>
</tr>
<tr>
<td>disposition control</td>
<td>&lt;determined by record test or group test results&gt;</td>
</tr>
<tr>
<td>disposition when validity test true</td>
<td>&lt;deferred&gt;</td>
</tr>
<tr>
<td>disposition when validity test false</td>
<td>&lt;write to invalid node file&gt;</td>
</tr>
</tbody>
</table>

Figure 1: A Hypothetical System/User Interaction Based on portions of the Validate Template
Figure 2: Sample Generic Code and Skeletal Source Code for the Validate Template.
1.3 Survey of Related Research

Capturing programming experience in the file/database problem domain, then organizing and presenting this experience in the form of templates, relies on a variety of research efforts aimed at improving software development. Each of these areas are surveyed by summarizing the contributions upon which this research was founded.

This survey is organized as follows:

- The use and importance of conceptual models.
- The role of human factors research in formation of conceptual models.
- Programming tools whose basis is reusable program components.
- The artificial intelligence approach to programming environments.
- The impact of database management systems and descriptive programming languages.
- Grammar theory used to represent template definitions.

1.3.1 Conceptual Models Applicable to the Domain of Study

As stated in the introduction, a conceptual model appropriate for the problem domain was needed. Research techniques and categorization of findings would then follow this conceptual model.

Proponents of software design and development methodologies must also first formulate a model to serve as the basis of conceptual integrity for the methodology. Brooks found conceptual
integrity to be the most important consideration in system design [Broo75].

1.3.1.1 Jackson and Warnier Methodologies

Methodologies such as Jackson [Jack75] and Warnier [Warn74] use tree-like diagrams to describe the structure present in the inputs and outputs to a program. The conceptual model underlying both of these methodologies simply stated is "the structure present in the data dictates the control structure of the program." The programmer, after creating data structure diagrams which describe the structure present in individual data inputs and outputs, must consolidate the individual data structure diagrams into a single program structure diagram. This is accomplished by identifying correspondences among nodes of the data structure diagrams. These correspondences can be 1:1, allowing for a combination of several nodes into a single node of the program structure diagram. Alternatively, one node may be hierarchically related to others, i.e., 1:N, in which case, several nodes are made subordinate to a single node. The nodes may be related by a time sequence which results in one node being placed before another in sequence. When such correspondences cannot be found even after redefinition of one or more of the data structure diagrams, additional techniques, namely "backtracking", resolution of "structure clashes" by sorting or "program inversion" are employed.
In defining templates, the author relied heavily on the correspondence between the structure of the data present in files/databases and the control code needed in the algorithm, both to traverse the data structures and to construct data aggregates not readily available in storage structures. It should be noted that both Jackson and Warnier methodologies emphasize control code and leave the programmer with little guidance as to the location or content of process code needed to manipulate data.

Definition: Control Code - those programming language statements in an algorithm that control the flow of execution and provide program branches in which process code can be placed.

Definition: Process Code - those programming language statements in an algorithm that provide data manipulation functions.

Since these methodologies approach each set of requirements without regard to the typical input/output structures present in a class of problems, the data and program structure diagrams of commonly occurring problems must be developed from scratch for every application. While these methodologies do discipline the designer, they in no way guarantee a unique solution.
1.3.1.2 Algorithms + Data Structures = Programs

In his classic text, *Algorithms + Data Structures = Programs*, Wirth recognizes a similar relationship between algorithms and data structures, but he does not give data structures prominence over the algorithms [Wirt76]. He sees the two as being "inseparably intertwined." He does, however, advocate the use of sample techniques to crystallize elementary composition principles, i.e., pattern algorithms with corresponding data structures. As with the previous two methodologies, Wirth's model ignores the semantics in classes of problems in order to concentrate on abstract structures and the corresponding abstract procedures. The programmer thus has a set of techniques which he must fit to the semantics of the application.

1.3.1.3 Algorithms = Logic + Control

Kowalski finds fault with the data structure school of methodologies [Kowa79]. He offers the model "Algorithms = Logic + Control", as an alternative. This model is not descriptive of what he has observed but, if followed, would offer improvements to the development process. His logic component specifies the problem domain-specific aspects of an application. It defines the problem in terms of what is to be accomplished. His control component defines the problem-solving strategy by which knowledge of the problem-solving domain is used to reach a solution; in this case, an
algorithm. In using this model to analyze algorithms, he finds that several interpretations of what constitutes control and what constitutes logic are possible.

He argues for the separation of the logic aspect of an algorithm from the control aspect as a way of improving algorithm structure and efficiency. This categorization also suggests an explanation of the non-uniqueness of solutions to a problem. He identifies algorithms as being equivalent if they solve the same problem with the same result, i.e., the logic components of the equivalent algorithms are identical, but the control components differ. A change in control affects the structure and efficiency of the algorithm, but still solves the same problem, provided the logic aspects remain the same.

In the template approach to algorithm formation given in this thesis, problem semantics (i.e., his logic component) first are captured and dealt with apart from specific data structures or programming language details. However, in order to generate skeletal code solutions, the problem semantics are combined with programming language details during the encoding of the template definition.

A degree of separation (logic from control in his terms) is still present in the template approach since the programmer may query the derivation tree constructed by the template system for specific problem domain semantics apart from implementation details.
Further, if the one who is encoding a template definition wishes to change the "control" aspects of a template, the problem domain semantics do not change; however, he may change the order in which they are considered, the interrelationships of semantic concepts and the semantic functions which generate code, while defining a new attributed grammar form for the template.

In summation, existing research points to the need for a conceptual model which incorporates the relationship between the structures present in files/databases and the structure of control code, but also takes advantage of semantics present in problem classes. These semantics describe the use of file/database structures (purpose, number and relationships), as well as the types and locations of process code within the control framework.

1.3.2 Human Factors Experimentation

Two popular research techniques available for determining what would improve the software development and maintenance process are:

- case studies -- which avoid experimental bias but are necessarily conducted in an uncontrolled environment, and

- controlled experiments -- which require the design of an experiment which is then conducted with subjects under controlled conditions.

Two controlled experiments were conducted in the early stages of this reported research in order to gain familiarity with a preliminary version of the conceptual model. Both are reported in
Chapter 2. One experiment did yield statistically significant results showing that programmers would choose a variety of correct control structures to solve a problem. The second experiment failed to produce statistically significant results, but did result in refinement of the conceptual model problem domain semantics. These two experiments, the need to identify domain semantics, and a number of articles which question the results, as well as the experimental methodology of recently reported experiments, led to the selection of the case study approach as the main research vehicle. Briefly, here are the concerns raised by researchers with respect to controlled experimentation as applied in software research.

Sheil finds a "sad state" existing in empirical studies [Shei81]. Because of the need for efficiency in the experiments, the following faults exist:

1. Experimental treatments are applied in short training sessions which he finds unreasonable.

2. "The point of an experiment is to generalize to some other (real world) situation. ... Consequently, one's experimental treatments should, in some sense, be representative of that other situation both in type and in strength." This point is frequently missing in existing experiments.

3. "Slight systematic differences between conditions tend to be washed out by large within-condition differences."

4. "... the difficulty of programming is a very nonlinear function of the size of the problem" which is often not taken into account in experimental design.

5. "Programming is clearly a learned skill, and, therefore, what is easy or difficult is much more a function of what
skills an individual has learned than of any inherent quality of the task."

Similar concerns are voiced by Brooks [Broo77], [Broo80] when he reports, "Even among programmers of very similar experience levels, differences of as much as 100 to 1 were found across programmers in the time taken to write a given program. ... Additionally, across problems constructed to be of similar difficulty, an individual programmer often displayed a six-fold difference in writing time." Because of the complexity of experiments, students are often employed as subjects, but Brooks questions the validity of generalizing student results to practicing programmers.

1.3.3 Reusable Software Components

Because templates, which are an enhancement of the concept of reusable components, form the foundation of this thesis, a review of research in this category is critical and will appear in depth. Three categories of references concerning reusable components exist in the research: 1) the identification of reusable components as a major approach to improving software development; 2) the role of components in explaining or modeling the behavior of programmers, and 3) the attempt to define or use components for software development within the file/database problem domain.
1.3.3.1 Identification of Reusable Components

This category is concerned with the call for research in the area of reusable components. The references presented come from well-respected, well-known sources and have occurred over a number of years which perhaps shows the lack of progress that has been made. Wasserman and Belady [Wass78] saw three paramount issues in software engineering as:

1. Software componentry — how to identify and define software components and develop a methodology for integrating them into systems,

2. The need to develop means to capitalize on prior programming experience, and

3. Enhancement of means of technology transfer between academia and industry.

An inventory of verified, easily modified, and reusable components is necessary before a better balance can be struck between free innovation and a disciplined evolution of programs. Wasserman and Belady's identify problems in constructing an inventory of components, which are:

1. Find the hidden commonality of functions across many applications requiring study of a large amount of complex and obscure software.

2. Experiment to determine which of the many alternative implementations of a function is preferred.

3. Determine appropriate component size.

4. Develop techniques for abstractly describing software components.
In regard to their third point, the author found useful templates to vary widely in size. There was no optimum size that a template need fit. Winograd looked to move the programmer away from having to describe an algorithm as a detailed set of instructions toward the interconnection of building blocks constructed by supplying parameters to prototypes for the instantiation of application specific building blocks or components [Wino79]. The building blocks must be \textit{flexible} so that they can grow to \textit{fit an environment of need} rather than a single well-specified task. Lack of flexibility and difficulties of using existing packages often make it easier to repeat the implementation of the function rather than integrate an existing package. Winograd saw a need to identify prototypes for describing those things which are \textit{common} to all of our programs.

Parnas called for the establishment of families of programs. A set of programs constitutes a family whenever it is worthwhile to study a program from the set by first studying common properties of the set and then determining the special properties of the individual program [Parn76]. He then discussed \textit{classical program development} termed "sequential completion" wherein new programs in a family are developed by modification of a complete working program. Such a procedure forces the programmer to recognize and categorize appropriate and inappropriate portions of the working program. The inappropriate portions are then eliminated, although removal of all
effects of these inappropriate portions may be impossible. These
difficulties just discussed point to the need for newer techniques
that maintain precise representations of intermediate versions of
programs as a starting point for proper modification.

The templates developed as part of this research contain the
semantics for a family of programs. The attributed grammars used to
encode templates are a precise representation. When a template is
applied to a problem, a derivation tree is built which is a precise
intermediate representation of the solution to a problem that
records design decisions. While it is intended for the user to
begin at the template stage for each new problem, it is possible to
have common variations of templates already defined. The one
defining the template accomplishes this by supplying default answers
to some of the calls for information (prompts) generated by the
template. Thus, with the template approach, several levels of
precise intermediate representations are possible.

Tracing the literature on one well-known pattern of programming
within the file/database domain -- the update of a sequential file,
this author saw emphasis on implementation technique rather than on
the need for a complete understanding and definition of the
semantics of the pattern. Dijkstra described an algorithm
previously invented by W. H. J. Feigen to solve a specific variety
of the update problem, but he was not concerned with defining the
complete semantics of the update problem class [Dijk76]. Other
authors Dwyer [Dwye81], Inglis [Ingl81], and Levy [Levy82] described and discussed alternative design and implementation approaches. While semantics were discussed to a limited extent, the emphasis of the articles is not problem class semantics, but the use of different development approaches, such as abstract data types. A major contribution of the research in this dissertation is the identification and organization of domain semantics quite separate from the implementation techniques used to realize solutions.

1.3.3.2 Role of Components in Software Psychology

The second area of research in which components are frequently used is software psychology. In this area, components are used in descriptive models of programmer behavior. Many software psychologists have used components to successfully describe the behavioral patterns of programmers. The conceptual organization imposed by the programmer was seen by Brooks as an important element in determining how easy a program is to write or understand [Broo77]. This further suggests that the size and kind of conceptual units a programmer has available will be an important determinant of his programming behavior. Performance is enhanced if the programmer has at hand an appropriate set of rules. However, this usually occurs only if the problem is of a type well-known to the programmer.

Sheil shared the notion that programming knowledge can be
thought of as a collection of facts, frames, paradigms, or schematics, each of which is organized as a program fragment, abstract to some degree, together with a set of propositions about their behavior and rules for combining them with others [Shei81]. This knowledge is indexed in terms of problem classes for which the component is appropriate.

In Shneiderman's syntactic/semantic model of the programming process, he described an experienced programmer as possessing a complex multilevel body of knowledge stored in long term memory and consisting of concepts and techniques [Shne80]. He also described a process very similar to that of instantiation of templates wherein specific problem details and a programmer's general knowledge from long term memory (i.e., his patterns) come together in working memory to formulate a solution.

1.3.3.3 Existing Efforts to Utilize Components in Program Development

The third category of reusable component research consists of efforts which utilize components in the development process. These efforts all involve some form of implementation, i.e., automating of the development process based on components through use of tool(s). The systems differ widely on the degree of automation, the sophistication and level of components used and the nature of reliance on the programmer. Methodologies that do not rely to some
extent on automated support are not included in this review.

ROGER'S METHODOLOGY

Rogers defined a simple architecture for consistent design within individual application programs [Roge83]. The architecture is based on four distinct levels within a program:

- **iteration** — consisting of initial and final processes and program operation conditions,
- **decision logic** — identifying the type of application program and dealing with major program logic flow decisions,
- **detailed processing** — constructing data aggregates and data manipulation operations
- **utility functions** — providing common functions usable by all levels in the program hierarchy.

He noted the narrow focus of earlier efforts at reusable components, i.e., the inability to handle the different elements and data structures that occur in various application requirements. Lacking an ability within his system to discover this variability and respond with an automatic code generation feature, he chose to represent his reusable components in the form of pseudocode modules, which, because of the abstract nature of pseudocode, provides a measure of flexibility. The burden of source code generation (i.e., instantiation of the pseudocode) is placed on the application programmer. Flexibility remains compromised in his approach as can be seen by the four types of decision logic included: control break (traversing a nested hierarchical relation defined by key field
sequences within a single source), balanced line (matching of sets of streams of sequential items), single-panel interactive, multi-panel interactive. It appeared from examples in his article that, his pseudocode models did contain a significant amount of problem semantics in the form of narrative annotations, i.e., a help facility. He did not appear to translate much of the semantics into actual code or program structure. While he mentioned over half of the templates included in the Enhanced Template Set defined in this dissertation, he was not able to address the variety of application requirements that the Enhanced Template Set does. This was because of his choice of pseudocode to represent his templates, the limited decision logic provided and the fact that his methodology is to be used only within a single program not in any other combinations.

**THE WESTINGHOUSE PDM METHODOLOGY**

An effort was initiated in 1976, by the Management Systems and Services Division of Westinghouse's Telecomputer Center in Pittsburgh, Pennsylvania to standardize the development of data processing programs [West79]. Based on previous data processing experience, nine programs and a methodology for using them were defined. These were called the Program Development Methodology (PDM). Each model program, defined as a macro, consisted of skeletal control code with documentation indicating how the skeleton was to be completed. In two years, approximately 180 programs
(which was 100% of the division's new programs) were developed using these skeletons. It was found that the use of program skeletons with embedded comments which described how to supply the details regarding a specific application, in fact, reduced program development costs in the long run. Of course, there was an initial cost overhead involved in teaching the programmers the use of the skeletons. The skeletons were limited in their ability to be combined with other skeleton programs and in the amount of problem domain semantics that were incorporated. While they generated high-level code to open, access and close files, as well as the initial loops needed to traverse simple relationships present in the files, little or no aid in forming process code was present because of the lack of problem class semantics. Programmers frequently had to modify automatically-generated code to allow for template interaction and combination. Combination of skeleton programs was done by the use of a sort procedure wherein one template would be executed placing its output in a file that was then sorted. The sorted file then became input to the next skeleton program.

The research of this thesis on isolating "fundamental" program development patterns was initially based on the Westinghouse experience summarized above. To judge the extent of semantics present in the Westinghouse PDM model programs, each of them was rated using the semantics present in the Enhanced Template Set. The results of that rating appear in Appendix E.
THE CSMAGIC PROGRAM GENERATION TOOL

CSMAGIC described by Bishop and Grace [Bish80] is a model-program based system which interactively collects detailed specification data from the programmer and automatically generates Cobol source code. CSMAGIC emphasizes the automation of five basic program types (i.e., pattern programs) which are data entry (including data validation), inquiry, update, print (report generation), and deletion (a variation of update). The template set defined in this dissertation is more complete and contains templates at the level of the CSMAGIC program types, but also at higher, as well as lower levels. The semantics of their program types are limited as seen by the small number of validity test types allowed, and the restriction that update must be done to an indexed file.

CSMAGIC contains a module for each of the five program types which consist of a model program (skeletal Cobol source code) and a program to manipulate the model during instantiation. This model-manipulation program serves the same function as "plans" in the system architecture described in this thesis. Its manipulations of the model program are limited to:

1. Copying a source line intact from the model.
2. Replacing a model source line (prompt) with the user's response.
3. Repeating a model source line.

Because of the inability to describe combinations of model
source lines, CSMAGIC is limited in its ability to respond to variety in application specifications.

CHEATHAM'S PROGRAM DEVELOPMENT SYSTEM

Cheatham described the Program Development System (PDS) which is a generic (independent of a particular methodology or problem domain semantics) programming environment that provides integrated tool support for module description, consistency checking, version control and refinement of modules [Chea79]. Unlike systems reviewed to this point, PDS does not identify a library of reusable components, but the rewrite facility of PDS could provide the basis for using a collection of templates defined for a problem domain. The rewrite facility allows user-defined replacement or transformations to take place on abstract modules, thereby producing a more concrete module. The rewrite facility is applied automatically by the system and operates by matching patterns. Upon finding a match the transformation is applied. These rewrite rules may be accompanied by predicates which govern their execution. In the rule-based automatic execution of transformations, PDS resembles an artificial intelligence system.

CORNELL PROGRAM SYNTHESIZER

The Cornell Program Synthesizer (CPS) is an interactive programming environment with facilities to create, edit, execute and debug programs [Teit81b]. The aim of this system is more toward
coding and debugging than any of the earlier systems discussed. Instead of coding at the character level, CPS provides syntactic units called templates (not to be confused with the use of templates in this thesis). CPS templates deal with the syntax of a particular programming language relying on the user to identify the need for a syntactic template as well as construction of assignment statements and expressions within a syntactic template. CPS does not encode problem domain semantics nor aid in collecting application specifications.

1.3.4 Artificial Intelligence Approach to Program Environments

Artificial intelligence (AI) is that branch of computer science that attempts to have machines emulate intelligent behavior. While AI techniques have proven valuable for some applications (e.g., medical diagnosis), the same techniques have yet to produce practical results in program development. To be successful, the AI approach has to meet the following criteria:

- the ability to perform the task,
- the ability to apply decision making,
- the ability to encode the above in machine processable form.

Perhaps the problem with AI being used in program development is the failure to fulfill these three criteria. A good knowledge base describing how to perform program development has not yet been
constructed, and inference rules to be used in decision making have not been formulated. Program transformation systems [Part83] are a notable AI effort at program generation.

Transformational Implementation (TI) is one of these program transformation systems. TI as described by Balzer [Balz79] seeks to automate implementation of programmer-selected transformations which operate on abstract program specifications to yield a more concrete optimized program. Transforms include conditions under which they are equivalence preserving. The system checks these conditions before applying the transformation, thus ensuring a correct transformation. Optimization to control code such as merging consecutive loops into a single loop and reducing IF-THEN sequences through the use of logical connections (AND, OR, NOT) are examples of transformations.

A second dimension of the transformational implementation approach is the transformation of abstract data structures into equivalent concrete structures. This is done by substituting the definition of accesses to the abstract objects for accesses to the concrete object.

The research presented in this thesis confirms that the use of manual optimization techniques contribute to many of the problems of software development and maintenance. Further, these optimization techniques often obscure the commonality in programs, making the identification of reusable components a difficult task.
Transformational implementation techniques for optimizing control code would merge nicely with the findings of this thesis by providing a mechanism for optimizing a template generated solution.

1.3.5 Database Management Systems Software and Descriptive Languages

Representing relationships between records and traversing those relationships, both for the purpose of creating new relationships and for processing data to produce information, is a key concern of the file/database problem domain. Traversal of relationships represented in storage structures is simpler than constructing relationships by accessing records residing in one or more unrelated storage structures.

Traditional sequential and indexed sequential file systems are limited in the variety and number of relationships that they can represent in a storage structure at any one time. Therefore, as file processing applications have called for the use of more and more files and the construction of more complex relationships, the control code in the resultant algorithms has likewise become more complex. Further problems with the use of traditional file systems that have motivated the development of database management systems are:

1. The definitions of the files and relationships are contained in (owned) by the individual application programs and thus are not readily available to serve other application needs.
2. Because files are optimized to specific program needs, their data is not readily available or in the form needed by other programs. This leads to formation of separate files containing redundant data.

3. Where two or more programs shares a file, changes in the file to respond to maintenance requests for one program necessitates maintenance on other programs even if the request has no meaning for other applications.

4. These and other reasons lead to data integrity problems, unreliable data, systems that are unresponsive to user requests for change, and data security concerns.

This state of affairs, plus the realization that data is a valuable resource requiring centralized management has led to the development of database management systems (DBMS). A DBMS is an integrated, shared data facility whose definition and access is controlled by a software package. Numerous commercially available packages are in widespread use; for example, IMS [IBM], IDS II [Hone80], DMSII [Burr74], INGRES [Held75].

DBMS's are able to represent in storage structures many more relationships than traditional file systems. While more relationships can be represented, facilities such as subschemas and retrieval conditions attached to input verbs used in the program to access data from the database can shield the application programmer from knowledge of unwanted relationships.

Most DBMS's, whether they adopt the hierarchical, network (i.e., CODASYL), or relational model, provide the programmer two classes of languages for interacting with the database. These are commonly referred to as the Data Manipulation Language (DML) which
is procedural in nature and the query language which is descriptive
in nature.

1.3.5.1 Data Manipulation Language Facilities of DBMS

The DML is usually a collection of verbs which augment the
existing verbs in a high-level programming language such as Cobol,
Fortran, Pascal or PL/1 and provide the programmer with a means for
interfacing with the DBMS. DML verbs embedded in the host
programming language serve to open, close, and provide input/output
access functions for the database. Because of the nature of a
database (coexistence of many relationships with many record types
possible), DML verbs have powerful constructs (such as the segment
search augments SSA of IMS) for selecting records by record type, by
relation, and/or by specific conditions involving both key and non-
key fields. Through the use of these conditions, the programmer in
effect further customizes the stored relations to his needs, thus
simplifying the control code of the program.

The effect of DBMS's on programming patterns has been to
simplify the control code of the algorithms and to add many verbs to
be used in input and output operations. But because it is
impractical to represent all needed relationships as storage
structures and because more complex relationships are now being
processed, application programs written with embedded DML commands
remain complex. The classes of problems within the problem domain
have not changed significantly, i.e., the same set of characteristics and semantics are present. The DML has served to extend capabilities to more complex relationships by simplifying the traversal of the stored relations. Of course, since more relationships can be represented in storage structures, the need to construct relationships through control code in the application program has lessened.

1.3.5.2 Query Language Facilities and Fourth Generation Languages

The second major advance fueled by the move to DBMS's is the advent of end-user, high-level languages for the purpose of querying the database. These query languages are descriptive not procedural in nature, and thus they appeal to end-users because "programmers" may be circumvented. Current query languages such as QBE [Zloo77], SQL [Astr75], IDP [Hone80], were founded upon early advances with descriptive languages such as RPG and the Report Writer module of Cobol. For a good description of motivations and the evolution of DBMS Systems and query languages, see Fry [Fry76] or Date [Date81].

Query languages, as indicated by the name, were initially designed to permit fast response to user generated, ad hoc queries for retrieval of data. While many query languages contain update and validation constructs, severe restrictions exist on their application. One such restriction is that the user must have knowledge of the entire storage structure associated with a record
or relationship before an update can be successful. Such knowledge usually resides with a programmer, not the user. Therefore, use of query languages is mostly restricted to simple ad hoc queries

A partial response to the limitations of existing query languages is the development of so called fourth generation languages, such as Focus and Mantis. These represent the next generation of descriptive languages which are either DBMS's in their own right or are interfaced with existing DBMS's.

FOCUS is the most popular and widely used of the fourth generation languages. Even so, only a small amount of material beyond the vendor supplied manuals has been written about it. Because FOCUS is typical of the new descriptive languages (systems), its features, limitations, and possible impact on the file/database problem domain will be reviewed.

FOCUS is best described as a DBMS which provides a full range of data processing functions for FOCUS files and a subset of those same functions, through an interface facility, for traditional files (QSAM, VSAM, ISAM) and other popular DBMS (IMS, IDMS, TOTAL and ADABASE).

The capabilities of the FOCUS file system were inspired by the existing DBMS technology in the sense that file descriptions are captured and stored apart from the application program in a facility resembling a data dictionary. These file/data definitions use a
syntax very closely related to the syntax of the Data Base Description Language (DBD) of IBM's IMS database system. Records called segments (note the IMS terminology) are related using the parent-descendant mechanism resulting in the multilevel tree structure of the hierarchical model. "Virtual" segments in FOCUS allow segments not stored in one tree to be accessed from other trees, similar to the IMS logical database. Indexing provides alternate access paths to segments.

Much of this logical and physical structure of data is not visible to the user of FOCUS because of the non-procedural language that unites all of the FOCUS facilities. Functions are carried out with a handful of English-like statements that replace thousands of lines of Cobol or PL/1 program code. The entire FOCUS vocabulary is under 100 words. On-line error correction and help functions permit immediate correction of mistakes and resumption of processing from the point of the error.

For report purposes only, FOCUS can process non-FOCUS files/databases as long as the file/database has been described to FOCUS in the DBD language. FOCUS can produce temporary working files through its HOLD, SAVE, and SAVB functions. These working files are just detail lines of reports that FOCUS can generate; i.e., reports less the headings, footings, and summary data. These features offer compatibility with a non-FOCUS environment, as well as a means of overcoming (by switching to another language, i.e.,
Cobol) syntactic restrictions of the FOCUS query language.

The query language of FOCUS resembles a relational algebra, particularly its MATCH feature. MATCH could be described as combining the projection and universal join operations of relational algebra.

FOCUS has a host language interface for accessing FOCUS files from traditional programming languages. The purpose and syntax of this interface were not presented in the documentation. Perhaps this is provided to give compatibility. Perhaps there are needs that FOCUS does not meet.

The data definition/data dictionary facility is a major simplifying force in FOCUS. It serves as the source of defaults in defining attributes for the data. For example, the display format of the field, and the field names or aliases to be used as column headers on reports are stored there. This can provide a prototype capability, i.e., generate a report using default field names as titles, then, when the report satisfies the user replace the field names with more descriptive titles, thus producing a finished report.

A complete, detail/summary report generation capability is provided using multilevel control breaks including subtotals, grand totals, row and column totals, and sorting by a combination of keys. Exercising the full power of this feature is not simple, but it is certainly simpler than generating the same reports by the
traditional procedure-oriented high-level programming languages.

Research on the impact of fourth generation languages in general and FOCUS in particular is almost nonexistent. In what has been written, users report that utilization of computer resources is much higher with FOCUS than procedural programs, and that computations are limited in FOCUS. If application requirements are simple, the end-user acceptance is possible, but when the requirements are more complex and require programmer-like knowledge and experience, end-user acceptance is less likely.

Chapin reported the results of a preliminary study of the impact of fourth generation languages on program maintenance [Chap84]. Because of the unscientific nature of the survey and the small number of returns, it would be incorrect to judge the findings as conclusive, but the reported observations point to difficulty in maintaining software coded in a fourth generation language. Comments range from "noxious in maintenance" to "usually faster, sometimes easier, but rarely cheaper." The size of the program apparently is critical. When the program module is less than 45 lines of source code, many of the maintenance complaints are alleviated. For longer, monolithic software (100 to several thousands lines) the use of FOCUS may make maintenance more expensive, which is not a desirable result.

In summary, the semantics of the file/database problem domain are not altered either by DBMS or fourth generation descriptive
languages. Database management systems allow for a larger number and a wider variety of relationships to be represented in storage structures. The DML verbs permit, to some extent, the programmer to receive only the records that are needed. But the relationships present in the database must still be traversed and others constructed. Process code is still nested in the control structures.

Because of current limitations and the difficulty of constructing complex queries, it is believed that fourth generation languages will not replace procedural language programs as the most common method of program construction within the near future, but it is too early to tell. Since the semantics of the file/database problem domain do not appear to be affected, but only the implementation of the algorithm, templates could be constructed to generate fourth generation language code by changing the code generation functions present in template definition.

1.3.6 Grammar Theory and Templates

A powerful formalism is needed to encode the definition of templates because of the large number of details representing structural (as well as semantic) concepts, and because of the complex combinations possible. Two techniques used in other component systems, namely use of pseudocode and use of generic program fragments written in a high-level programming language, are
limited in their ability to deal with the variety of combinations of semantics found in problem classes. Systems using these approaches either limit the semantic content, as was the case with the Westinghouse PDM model program library [West79], or they rely on a large number of components, each of which deals with a specific set of application requirements, as does the approach adopted by Rogers [Roge83].

However, context-free grammars are a natural, efficient way to represent the complex syntactic details of programming languages. Grammars, therefore, offer a hope of overcoming the inadequacies of both the pseudocode and generic programming language approaches to encoding components.

Context-free grammars do not adequately deal with semantic information, which is of primary importance in encoding template definitions. A similar concern for semantics and context-free grammars led Knuth to pioneer research in attributed grammars [Knut68]. Attributed grammars allow for the generation of the meaning of strings in the language, as well as representing the syntax of the language. This is accomplished through the use of attributes associated with the symbols of the vocabulary of the grammar and semantic functions associated with productions of the grammar. Attributes represent important characteristics, often context-sensitive information, which is not handled by the context-free grammar. Parameters used by the semantic functions may be
attribute values passed down the derivation tree, "i.e., inherited attributes," or attribute values passed up the derivation tree, "i.e., synthesized attributes." Evaluating a semantic function causes the value of the attribute in question to be determined. The generation of object code by a compiler is a good example of the use of semantic functions.

In the template-based approach to program development, the attributes record development decisions and important program information. The research of this thesis identified limitations of traditional semantic functions which suggests that an extension of semantic functions is needed as further work. One step in this direction is the a plans defined in templates. Plans still evaluate attributes, but they can also manipulate the template definition which is not a traditional role for semantic functions. One type of manipulation carried out by "plans" is to tune the definition of the template to better match the specific application being worked on. This tuning could take the form of eliminating unneeded productions or modifying productions to eliminate unneeded terminal and/or nonterminal symbols.

A notable effort which used attributed grammars to produce code for a restricted problem class was reported by Logrippo and Skuce [Logr83]. Their work is of further interest because of the use of the structure present in an input file to generate a skeletal program in a high-level language. Because the use of attributed
grammars and generated control code based on the structure of the input file are closely related to the research reported here, the work of Logrippo and Skuce will be described in-depth in a later chapter.

One final capability is needed to model templates. This is the task of representing the similarities (structural and semantic) existing within a problem class and among different versions (because of tuning) of a single template. Grammar forms had been used by Ginsberg to represent similarity among grammars [Gins76]. A grammar form represents a family of grammars and, under a systematic mapping, the grammar form could be used to generate any member of the family. This is possible because of similarities among family members, such as the shape of the derivation tree, languages generated, and processing complexity.

The grammar form contains a master or representative grammar which, when transformed, can produce any of the grammars in the family through a process called interpretation. In the interpretation process, a mapping is applied to the vocabulary and productions of the master grammar which form result in an interpreted grammar. This transformation can be an iterative process with the resulting (interpreted) grammar taking the role of the master grammar for a subclass of the original family.

This capability is useful in two ways: to derive the definitions of new templates from old templates, and to derive the
program by user provided instantiation of prompts. Both of these uses will be demonstrated in later chapters.

In a grammar form, the representative grammar is a context-free grammar. Based on the preceding intuitive discussion, what needed in the template approach is a grammar form where the representative grammar is an attributed context-free grammar. The attributed grammar form model, as defined by Soni, results from a synthesis of these ideas [Soni83a]. A definition borrowed from Kuo [Kuo83c] is given in Appendix F. As an example, the attributed grammar form model is used to develop the definition of two templates in Chapter 5. During development of the example, restrictions present in the model are identified, which suggests the model might have to be further developed.

1.4 Scope of the Thesis and Contributions

The contributions of this thesis fall into three categories.

1. The development and definition of a conceptual model which is descriptive of the file/database problem domain.

2. The use of this conceptual model to discover and categorize problem-domain semantics by conducting an extensive case study of production programs.

3. The development of the template concept which, when encoded using attributed grammars, forms the basis of an automated software development environment.

Discovery of semantics within the problem domain requires the formation of a conceptual model that accurately describes problems
and solutions encountered within the domain and allows for categorizing findings. Chapter 2 contains the steps followed in forming a conceptual model for the file/database problem domain. This conceptual model is able to integrate the correspondence between data structures and control code while at the same time using semantics to capture commonality in control code and processing functions.

The conceptual model is the basis for defining a hypothesis template set as a first estimate of reusable components within the domain. An extensive case study of production software has been undertaken to establish the utility and content of the Hypothesis Template Set. This case study, which is reported in Chapters 3 and 4 and Appendices C and D, resulted in the following contributions:

- identification of patterns appearing in production programs with the definitions of terms and semantics to describe these patterns,
- expansion of existing templates to include newly discovered semantics, thus resulting in more realistic templates,
- definition of additional templates to be included in the set,
- establishment of the need for templates to be used in combination and the nature of the combinations,
- upgrading of confidence in the utility of the Enhanced Template Set because of the extensive, repeated use of template semantics within the production programs.

The Enhanced Template Set resulting from the case study was able to strike a balance between a semantically weak, widely
applicable methodology such as Jackson [Jack75], and existing attempts at reusable components which are semantically specific but severely restricted in scope of application. This is accomplished by organizing major variations of a class of problems into a single template, and then providing a means for both determining the variation needed and tuning the template for that variation.

The Enhanced Template Set serves both as a semantic characterization of the problem domain and as a knowledge base for software specification and development.

Because of the volume of information contained in the templates and the need to be user-friendly, a means for encoding and manipulating templates as part of an automated software environment had to be formulated. Attributed grammar forms are used in Chapter 5 as a formalism for encoding the templates. The types of transformations needed to define new templates based on existing templates are also developed in Chapter 5, along with an example of such a template definition. Chapter 6 then compares the source code produced by applying the Logrippo, Jackson, and template approaches to a set of "validate" problem specifications.

Chapter 7 consists of defining the architecture for a template based software environment and a scenario of the interaction between the system and user, and between components of the system. In the scenario, "plans" are invoked by the various "experts" defined in the architecture. The scenario also shows how the template
contributes to the creation of an application specific derivation

tree which serves as a concise intermediate form of the program.

The final chapter contains conclusions and identifies further
work to be accomplished.
Chapter 2

Formation of a Conceptual Model of the Problem Domain

2.1 Introduction

When dealing with a process as complex as software development, forming a model that describes relevant abstract concepts and relationships among those concepts is a critical first step toward understanding the process. Such a model provides a framework for recording and categorizing empirical observations. Further, the model can be the foundation upon which research techniques are designed and, thus, a thread for unifying the research. With this motivation, a search began for the components of such a model which would focus analysis of existing software in such a way as to permit identification of commonly occurring patterns and allow for categorizing these observations into reusable components.

The search for the model began by adopting the model that is the basis of both the Jackson [Jack75] and Warnier [Warn74] methodologies, both acclaimed as useful methodologies for the domain. The basis of their model is that a strong correspondence exists between the structures present in the inputs and outputs to an algorithm, and the structure of the algorithm. This model was
appropriate (at least as a start) because it addresses the locus of complexity within the domain, i.e., complex relations represented in the input/output structures and complex relations that need to be constructed by traversing the input structures (typically resident in secondary storage).

2.2 Preliminary Study of Fourteen Student Programs

It was felt that the initial model was too general in purpose. Therefore, a preliminary study of student programs was undertaken with the initial model used to suggest analysis techniques and to categorize resultant findings. It was hoped that problems with model validity and robustness (descriptive ability of the model) would surface along with ways to refine the model.

Fourteen PL/1 programs, written by undergraduate students in a sophomore level, techniques oriented file processing course were collected and studied. The students had received instruction in the Jackson methodology and were to apply the methodology in constructing a solution. Each student received an identical set of problem requirements dealing with the use of a master file to validate the data contained in a transaction file. Data from corresponding (matching key fields) transaction and master records were to be used in forming an expanded transaction record. Two reports were to be generated: an exception report listing any errors discovered and a summary report listing valid transactions.
A preliminary reading of the source code submitted by the students failed to show a common solution pattern in the algorithm as was naively expected, given the use of the same methodology on identical problem specifications. A large variety of correct ways of implementing traversals of the four data relationships, present either in storage structures or needing to be constructed, was actually reflected in these fourteen programs. The variety was no doubt due in part to the student's imperfect application of the methodology. A scheme was developed to categorize the control structures implemented in the programs. Two variables were found to describe the differences in control structure.

- Variable 1 described the technique used to define process branches either through nesting of one control structure within another, thus taking advantage of knowledge of the outer tested condition, or not nesting, which required retesting of conditions.

- Variable 2 dealt with the use made of a key-based grouping of records within the storage structure of one input file. Either a local loop which recognized the key grouping was present or no local loop was present and the relationship ignored.

The Nested condition value of variable 1 is represented in Figure 3, and the not nested, i.e., linear sequence of conditions, is represented in Figure 4. The second variable dealing with the local loop could be present in four forms: local loop in the match branch (see figure 5), local loop in the no match branch, local loop in both branches, or local loop not in either branch.

Table 1 shows the eight program structures possible from all
combinations of the two variable values, only four of which were observed in the fourteen program sample.

Variety in control structures alone did not account for all of the differences observed among the programs. There was also a great deal of difference in the process code used to implement data manipulation functions. In order to successfully realize a required process specification, the following must have occurred:

1. The user must have recognized and understood the specification;
2. The proper statements to implement the function must have been written; and

3. These statements must be placed or located in the proper branch(es) of the control code.

A large number of the programs did not successfully realize all the processes called for in the problem specifications. Table 2 summarizes our findings of the process functions in the student programs.
In the study, concentration centered on implementation and location errors. The errors that appeared to be caused by imprecise problem specifications were ignored. It is clear that a specification language that was understood by the programmer and that was descriptive of the concepts and dimensions of the problem class would have improved the performance of the students.

From the results in Table 2 it is evident that most implementation errors were caused by the omission of statements. Because in many cases the student was able to successfully implement
<table>
<thead>
<tr>
<th>Category</th>
<th>Nested</th>
<th>Local Loop in</th>
<th>Count in Category</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Not Matched</td>
<td>Matched</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>x</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Table 2: Implementation of Process Functions in Student Programs

<table>
<thead>
<tr>
<th>Process</th>
<th>Number of Programs Where the Process Was Successful</th>
<th>Number of Programs Where the Process Was Unsuccessful</th>
<th>Reasons for Failure of Process Implementation</th>
<th>Reasons for Failure of Process Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. VALIDATE Transaction Product Numbers</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2. VALIDATE Transaction Unit Prices</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3. Validate Transaction Quantity Fields</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4. Count of the Number of Valid Line Items</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5. Validate Transaction Dollar Amounts</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6. Construct and Output an Extended Record for Each Valid Transaction</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7. Construct and Output 2 Extended Records for Each Invalid Transaction</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8. Generate 1 Line of the Valid Transaction Report for Each Valid Transaction</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>9. Generate 1 Line of the Exception Report for Each Invalid Transaction</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10. Validate Transaction Quantity Fields by Error Type</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11. Count of the Number of Invalid Line Items by Error Type</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>12. Validate Transaction Dollar Amounts by Error Type</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>

**IMPLEMENTATION ERROR KEY**

1. Missing a required statement(s),
2. Using an incorrect operator(s) and/or operand within a statement,
3. Other.

**LOCATION ERROR KEY**

1. Process code located in the wrong branch,
2. Incorrect placement of process code in relation to a loop structure,
3. Proper process branch missing.
a similar process elsewhere, it was concluded that the reason for missing processes was failure to understand the problem specifications\textsuperscript{2}. Also, the table points out that students appear to make more errors in the placement of a process statement than in coding the statement. That is, even though the control structure in the program was correct, the process statements were placed incorrectly.

The program study demonstrated the ability of the conceptual model to categorize variety occurring within programs developed for a single set of specifications. Process code location (i.e. placement of statements within the control code) was a source of variety that could be categorized. Process code implementation (i.e. writing of the executable statements in the syntax of a programming language), while a source of variety, defied categorization. This latter point was due to the uniqueness of operand names, and the variety of ways in which operands were combined. The actual writing of the process code statement(s), which was not a major stumbling block and which, because of the uniqueness of such statements, could not easily become part of a reusable component, was therefore not incorporated into the model.

The conceptual model simply stated is:

\textsuperscript{2}In an informal followup questionnaire, students were able to write program language statements that correctly implemented isolated, specific problem specifications
1. Relationships present in the input and output structures can be used to determine the control structure of the algorithm.

2. Control code provides identifiable locations for the placement of programmer supplied process code.

This model thus allowed categorization along the dimensions of process code location and control code.

2.3 Controlled Experiments Based on Findings from the Preliminary Study

Programmers approach each programming problem with a collection of programming techniques gathered either through experience or education. They then employ these techniques as they design and construct programs. Methodologies such as Jackson, or Structured Design, attempt to discipline the programmer during the design process by focusing his attention on specific aspects of the programming problem, and by providing rules (some explicit and some more in the nature of heuristics) for making design decisions. The net effect of successfully applying a methodology is to restrict the possible number of designs to be considered. This channels the designer toward a final design which has many characteristics in common with those designs at which other programmers using the same methodology on the same application would have arrived. In other words, the algorithm is more standard, and patterns of programming should be visible.
The preliminary study of student programs showed that:

1. The control code and process code are two important aspects of algorithms in this domain.

2. A model based on these concepts is useful in categorizing programs in the domain.

3. A great deal of variety developed without the aid of an enforced methodology.

Two human factors experiments were designed in an attempt to duplicate or test the program study findings in a controlled environment. Specifically the experiments were designed to test if:

1. Programmers would choose a variety of correct control structures to solve a single set of problem specifications.

2. Familiarity with the pattern upon which the design of an algorithm was based would improve performance of maintenance tasks on the algorithm.

In the following sections, each experiment will be described in terms of the research and statistical hypotheses, the experimental method, the materials, the analysis, the results and the conclusions.

2.3.1 Experiment 1 - Variety in Control Structures

Based on variable 1 (nesting of conditions) and variable 2 (use of local loops), which described the variety found in the control structures of the student programs reported in the previous section, five control structures were developed which could successfully be used in implementing a solution to a set of problem specifications.
similar to those present in the student program study.

**Hypothesis**

**Research Hypothesis:**

Programmers will develop a variety of correct algorithms to solve a given set of specifications.

**Statistical Hypothesis:**

\[ H_0: F_1 = F_2 = F_3 = F_4 = F_5 \]

where \( F_i \) represents the frequency of an algorithm being selected as the one preferred by the programmer to solve the problem.

**Method**

The subjects consisted of 32 graduate and upper-level undergraduate computer science students enrolled in a dual-listed human factors course. The subjects were asked to study an extensive set of problem specifications written in free form. They were then asked to compare pairs of algorithms (available in both flowchart and pseudocode form), and state a preference for one algorithm over the other, a preference for both, or a preference for neither algorithm in the pair. The subjects were not instructed to apply any specific methodology in making their choices. The subjects were given as much time as they needed to study the specifications and compare the algorithms. Five algorithms suggested by the earlier student program study were used. Every algorithm was compared with
every other algorithm resulting in ten pairwise comparisons. The order or presentation of the pairs was randomized and the order of the algorithms within a pair was balanced.

Materials

A printed packet containing problem specifications; an instruction-answer sheet; a one page description of flowchart symbols; a two page description for each of the five algorithms that were numbered identically in all of the packets, one page in flowchart form and one page in pseudocode form, was given to the subjects (see Appendix A). The instruction-answer sheet randomly presented the pairs of algorithms to be considered. An area for circling the number of the preferred algorithm in the pair or 'N' which represented a dislike of both flowcharts in the pair was also present to the left of each pair of algorithm numbers. The subjects were not given any set of criteria upon which to base their preference.

Results

The number of times an algorithm number was circled on the answer sheet was counted and recorded for each subject. Then for each subject, the algorithm(s) with the largest number of circles was assumed to be the algorithm that would be selected by the subject to solve the problem. That algorithm was awarded one point for being selected. If several algorithms had an equal number of
circles and no algorithm had a higher number of circles, then each of the algorithms with the highest number of circles was awarded a proportional share of the selection point. For example, if subject seven circled algorithm numbers three and five three times each and all other algorithms were circled less than three times, then algorithms three and five were awarded one-half of a selection point. The selection points were then summed by algorithm. The results are shown in Table 3.

Table 3: Observed Frequency of Algorithm Selection in Experiment 1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observed Frequency</td>
<td>6.83</td>
<td>.83</td>
<td>8.33</td>
<td>8.5</td>
<td>6.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

One subject had marked "N" for all comparisons of algorithm pairs. In a post experiment interview, the subject indicated that he had not carefully considered the algorithms. Because of this, his results were excluded from analysis, although they do appear in
Table 3.

Analysis

The CHI-SQUARE test was used to see if the observed selections differed significantly from a random selection, i.e., one in which all the frequencies were equal.

$H_0$: $F_1=F_2=F_3=F_4=F_5$

with the decision rule to reject $H_0$ if

$$\text{CHI-SQUARE}_{\text{COMPUTED}} \geq \text{CHI-SQUARE}_{\text{TABLED}}$$

with alpha = .05 and df = 4

Table 4 shows the observed and expected frequencies of algorithm selection. The decision was to not reject the null hypothesis.

Discussion

The subjects had, as a result of experiment 1, selected a variety of control structures to solve the problem and the observations in the earlier study had been reinforced.

2.3.2 Experiment 2 — Effects of Pattern Familiarity on Performance of Maintenance Tasks

Recognition of a problem as one capable of being solved by a known solution technique or pattern has proven to be of great benefit in other disciplines. A pattern or known solution technique
Table 4: Observed and Expected Frequencies of Algorithm Selection

<table>
<thead>
<tr>
<th>ALGORITHM NUMBER</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observed frequency</td>
<td>6.83</td>
<td>.83</td>
<td>8.33</td>
<td>8.5</td>
<td>6.5</td>
</tr>
<tr>
<td>Expected frequency</td>
<td>6.2</td>
<td>6.2</td>
<td>6.2</td>
<td>6.2</td>
<td>6.2</td>
</tr>
</tbody>
</table>

where \( \frac{31}{5} = 6.2 \)

\[
\text{CHI-SQUARE}_{\text{COMPUTED}} = 6.35
\]

with alpha = .05 and df = 4, \( \text{CHI-SQUARE(TABLED)} = 9.488 \)

In software development includes the aspects of the conceptual model defined at the end of the study of the student programs. Therefore, knowledge of an appropriate control structure which traverses relationships existing in storage structures, constructs other needed relationships, provides locations to process records and data aggregates by creating branches within the control structure, should aid in software development and maintenance.

Experiment 2 sought to establish that familiarity with the
pattern upon which an algorithm was based would enhance the performance of maintenance tasks.

Hypothesis

Research Hypothesis:
Familiarity with the pattern used to develop a program would be beneficial in doing maintenance on the program.

Statistical Hypothesis:

\[ H_0: \text{mean}_{\text{familiar}} = \text{mean}_{\text{unfamiliar}} \]

i.e., the mean time to perform maintenance tasks on a program based on a familiar pattern when compared with the mean time to perform a like series of maintenance tasks on a program based on an unfamiliar pattern will statistically be the same.

Method

The subjects consisted of 12 graduate and upper-level undergraduate computer science students enrolled in a dual-listed human factors course. These 12 students were a subset of the students that participated in experiment 1. The remainder of the class participated in an initial version of this second experiment. The performance of the initial group was not reported because in post experiment interviews they reported ignoring the pattern while completing the assigned maintenance tasks on the familiar algorithm. The experiment design was altered to ensure use of the pattern and
it is this altered form of the experiment that is reported. The author also realized that a sample size of 12 could have negative effects on the results.

The experiment was a one-factor-within-subject design. This meant that data was collected from each subject under all the treatment levels of the independent variable (pattern familiarity) i.e., a subject would do maintenance on a program derived from a familiar, as well as an unfamiliar pattern. The order in which the familiar and unfamiliar programs were presented was balanced so that as many subjects would see the unfamiliar first followed by the familiar as would see the familiar first followed by the unfamiliar. The experiment consisted of nine components (the order of occurrence of these components will be discussed shortly):

1. Description of the problem domain.
2. Sample terminal dialog (unfamiliar first only).
3. Application and maintenance specifications for a business application.
5. Automated presentation and timing (dependent variable data collection) of business application maintenance task performance.
6. Tutorial on a pattern including terminal interaction for practice of the use of the pattern in maintenance.
7. Application and maintenance specifications for a voter application.
8. Voter application pseudocode listing.

Subjects were divided randomly into two groups defined by the order of the above tasks. One group, called "unfamiliar first" did the business application data collection step before any instruction about the pattern. This was followed by the voter application data collection. The other group called "familiar first," first learned the pattern, then did data collection using the voter application followed by data collection using the business application. Therefore, for the two groups, the sequence in which the above components were seen was:

Familiar first - 1, 6, 7, 8, 9, 3, 4, 5
Unfamiliar first - 1, 2, 3, 4, 5, 6, 7, 8, 9

The major change was that 6, 7, 8, 9 were taken out of sequence in the familiar first group. Also, the familiar first group did not need the sample terminal dialog (item 2) because they were given a sample of the dialog during the tutorial in which they learned about the pattern.

For both applications, the subjects were given two written pages of specifications for the application and a brief description of the maintenance tasks. They were told to study the specifications and were directed to enter "BEGIN" at the terminal when they were ready for data collection. Upon entering "BEGIN", a
program statement(s) was presented on the screen and timing began.

In the case of the business application algorithm which was always based on the unfamiliar pattern, whether it went first or second, the subjects were asked to study the application listing and enter a statement number from the listing indicating where the maintenance statement(s) in question should be placed. The program determined if the answer was correct or not. If it was not correct, the statement was redisplayed, a number-of-tries counter was incremented, and the experiment continued. If a correct answer was obtained, the programmer was presented with the next maintenance task. This continued until statements pertaining to all four maintenance requests were correctly placed.

The voter application algorithm was always based on the familiar pattern, i.e., the pattern just taught. While it is possible that a subject might be more familiar with one application area over the other, this was not considered to have a major impact because the problem specifications were simple and the subjects were given as much time as needed to study the specifications and ask questions before data collection began. For the voter application the students were presented with a maintenance statement or statement group, then instead of placing the statement in the application program, they were asked to supply a statement number from the pattern program, i.e., a program devoid of voter application terminology and based on the pattern algorithm.
If the statement number given by the subject for the pattern program was incorrect, the subject was told so. The maintenance statement was then redisplayed, and the subject was asked to try again with the pattern program.

If a correct answer was obtained in the pattern program, the subject was provided a pointer into the application program listing where that type of activity occurred. The programmer then checked the local context of the application program and supplied a second statement number for the placement of the statement, this time in the application program. This process continued until correct placement in the application program occurred. The experiment did not attempt to cause the programmer any difficulty in placing the statement in the application program once the area had been identified. There were cases where the local context had to be taken into account.

Materials

The materials consisted of a packet of papers containing items 1 through 9 from the above list in the appropriate order, and a program which generated the dialog and maintenance statements displayed at the terminal. A sample of the materials is present in Appendix B.
Results

The data organized by group with the time in seconds and the time and tries summarized over the four maintenance tasks is given in Tables 5 and 6, along with the mean and standard deviation for each column. Since a time stamp was placed on each user response and a log file was kept of the terminal interaction, corrections to the final data were possible. For example, one subject entered a comma when he wanted to enter a decimal point. The additional time and the attempt to correct the error were subtracted from the individual's total. Another individual discovered a correct location for a statement that was not recognized as being correct by the program. That score was also corrected.

Analysis

The data are reported in Tables 5 and 6. The time columns show the total completion time in seconds for all four maintenance tasks. The mean time for completing all four tasks ranges from a low of 450.0 seconds (7.5 minutes) to a high of 903.2 seconds (15.1 minutes). The times recorded consist of two components, those due to the independent variable and those incidental to the conduct of the experiment. Upon seeing the magnitude of the times an immediate concern was "effect size" due to the independent variable as compared with the incidental task times. The fear was that although the incidental tasks could be assumed to be constant within any one
### Table 5: Summary Data for the Familiar First Group

<table>
<thead>
<tr>
<th>Subject</th>
<th>Pattern Tries</th>
<th>Pattern Time</th>
<th>Application Tries</th>
<th>Application Time</th>
<th>Total Tries</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>4</td>
<td>317</td>
<td>4</td>
<td>173</td>
<td>8</td>
<td>490</td>
</tr>
<tr>
<td>34</td>
<td>6</td>
<td>496</td>
<td>4</td>
<td>99</td>
<td>10</td>
<td>595</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>542</td>
<td>4</td>
<td>237</td>
<td>10</td>
<td>779</td>
</tr>
<tr>
<td>33</td>
<td>12</td>
<td>648</td>
<td>4</td>
<td>135</td>
<td>16</td>
<td>783</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>1457</td>
<td>7</td>
<td>165</td>
<td>22</td>
<td>1622</td>
</tr>
<tr>
<td>19</td>
<td>6</td>
<td>458</td>
<td>4</td>
<td>159</td>
<td>10</td>
<td>617</td>
</tr>
<tr>
<td>mean</td>
<td>8.2</td>
<td>653.0</td>
<td>4.5</td>
<td>161.3</td>
<td>12.7</td>
<td>814.3</td>
</tr>
<tr>
<td>std dev</td>
<td>4.3</td>
<td>408.5</td>
<td>1.2</td>
<td>45.7</td>
<td>5.3</td>
<td>411.5</td>
</tr>
</tbody>
</table>

Programmer, they might well provide a majority of the time measured in the experiment. Such concerns were also voiced by Sheil in his critique of controlled experimentation [Shei81]. Clearly, in requiring the subjects to repeat these incidental tasks twice during the familiar phase of the experiment (once for the pattern program and once for the application algorithm), the effects of the
Table 6: Summary Data for the Unfamiliar First Group

<table>
<thead>
<tr>
<th>Subject</th>
<th>Pattern</th>
<th>Application</th>
<th>Total</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tries</td>
<td>Time</td>
<td>Tries</td>
<td>Time</td>
</tr>
<tr>
<td>27</td>
<td>6</td>
<td>220</td>
<td>4</td>
<td>97</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>193</td>
<td>5</td>
<td>66</td>
</tr>
<tr>
<td>29</td>
<td>11</td>
<td>663</td>
<td>6</td>
<td>153</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>407</td>
<td>5</td>
<td>131</td>
</tr>
<tr>
<td>30</td>
<td>12</td>
<td>1308</td>
<td>4</td>
<td>58</td>
</tr>
<tr>
<td>23</td>
<td>9</td>
<td>574</td>
<td>5</td>
<td>129</td>
</tr>
<tr>
<td>mean</td>
<td>7.7</td>
<td>560.8</td>
<td>4.8</td>
<td>105.7</td>
</tr>
<tr>
<td>std dev</td>
<td>3.5</td>
<td>410.9</td>
<td>.8</td>
<td>38.3</td>
</tr>
</tbody>
</table>

The experimental treatment could easily become insignificant. It was, therefore, decided to use only the pattern program times for analysis when considering the familiar phase of the experiment and compare these with the application program times recorded during the unfamiliar phase. Even so, as can be seen next, no statistically significant results caused by the experimental treatment were found.
Paired T-TESTs were executed to determine the following:

1. Was there any significant difference between the familiar pattern time for an individual and the unfamiliar application time regardless of order of presentation?

\[ H_0: \text{mean diff} = 0 \]

where "diff" is the difference between paired values for a subject. The paired T-TEST results are not significant with \( p = .65 \); therefore, the null hypothesis is not rejected.

2. Was there any significant difference between all first times and all second times regardless of familiarity?

\[ H_0: \text{mean diff} = 0 \]

where "diff" is the difference between a first time and a second time for each subject. The paired T-TEST results are not significant with \( p = .0542 \); therefore, the null hypothesis is not rejected.

T-TESTS (not paired T-TESTS are used since subject identity is lost) are executed to determine the following:

3. Did the mean time for familiar first differ significantly from unfamiliar first?

\[ H_0: \text{mean familiar first} = \text{mean unfamiliar first} \]

The results are not significant with \( p = .26 \), therefore the null hypothesis is not rejected.

4. Did the mean time for unfamiliar first differ significantly from unfamiliar second?

\[ H_0: \text{mean unfamiliar first} = \text{mean unfamiliar second} \]

The results indicated that the null hypothesis could be rejected with \( p = .013 \), a highly significant result.

5. There were no significant results when number-of-tries was used as the dependent variable.
Discussion

The results of this second experiment were disappointing. The possible reasons, which are related to the design of the experiment, will be briefly discussed.

1. A more complex program written in an existing high-level programming language should be used;

- The Cobol-like pseudocode used for the program listing was 40 executable lines long for application 1 and 46 lines for application 2.
- The pattern program for application 1 contained 35 lines of executable code.
- The smallness of the program code may have eliminated the location advantages provided by the pattern.

Solution: Use a program written in a current programming language known to the subjects instead of an abstraction of the code provided by the pseudocode.

2. Learning or carryover from the first data collection phase to the second data collection phase was excessive;

- Subjects appeared to get better at the experimental procedure.

Solution: prior to the experiment have the subjects practice the experimental procedure.

- The pattern included control structure and generic language. While the control structure was changed, the generic language, (i.e., problem class semantics), could be learned and used in the second data collection. Since knowledge of a problem probably includes an understanding in some terms of the processing needs and a control structure, the pattern must organize this knowledge well to be an effective aid.
Solution: Use a "between" group design instead of a "within" group design.

3. The independent variable in the experiment was pattern familiarity, but the tutorial meant to supply the treatment may not have been effective and certainly varied in the degree of effectiveness for individuals.

Solution:

- Teach the pattern in a series of classes (1-3).
  Give the subjects at least one post test. The post test should be scored to ascertain the level of familiarity attained by each student. For example, students scoring below 95% should be given additional training.

4. Doing maintenance on the pattern program compared with maintenance on the application code suffered from internal validity problems and should not be done.

5. In the familiar data collection phase subjects had to deal with a larger volume of material and an increase in manual tasks.

- There were 75 lines of pseudocode in the familiar phase versus 46 lines in the unfamiliar phase.

- The 75 lines of pseudocode in the familiar phase were at two different levels of abstraction (pattern level and application level).

- There were 5 additional lines of terminal output for a correct answer and 13 additional terminal output lines for an incorrect answer in the familiar phase.

- There were 8 pages of printed material for the familiar phase and 5 pages for the unfamiliar phase.

- The familiar tasks required paging through the 8 pages between the pattern program and the application program while the unfamiliar tasks could be completed by only looking at the application code.

- The effects of the familiarity treatment were too
small to override these differences in manual tasks.

Solution: Equalize the manual aspects of the procedure or remove time as the dependent variable.

2.4 Conclusions Drawn from the Two Experiments

In spite of these problems and the failure to obtain statistically significant results, this second experiment led to the realization that control code patterns with process code locations alone was not representative of the complexity of applications in the domain. The conceptual model, as yet, was not descriptive enough. This is due to a lack of complexity, which could have been one of the reasons for the experimental results. Further, the commonality in programs attributable to classes of application problems was not represented in the conceptual model. What was missing from the conceptual model was the semantics that are shared by programs from a problem class. These semantics provided understanding about the numbers, types, purposes and relationships of inputs and outputs for problems in the class. Further, knowledge and understanding about the data manipulation functions could also have been provided by these semantics. The conceptual model, if it were to accurately represent the experience present in the domain and be useful in the formation of reusable components of maximum utility, should have included problem class semantics.

The definition of the conceptual model of the domain was thus
refined to the following:

The file/database problem domain contains problem classes. The semantics of a problem class can be used in the following ways:

1. To identify the type, number, purpose and structure of typical inputs and outputs.

2. To produce control structures that would traverse the I/O structures, form needed data aggregates, and provide required process branches.

3. To identify the types and nature of data manipulations and, to some extent, produce process code.

Because of this new model it was no longer sufficient to note the correspondence between data structure and control structure and the process code locations present in the control code. To successfully capture patterns of programming, it was necessary to add a third dimension to the conceptual model. This third dimension deals with the semantics of classes of problems

- The types, numbers, purpose and common ways of combining records from data structures needed to be identified.

- Understanding of typically occurring processes, implementations of these processes and specific locations within the control code for those processes needed to be gathered.

A thorough understanding of problem classes within the domain was needed. In other words, the domain semantics organized into problem classes had to be identified. The means for accomplishing this goal was an extensive case study of production programs which is described in the next chapter.
Chapter 3

Case Study to Discover File/Database

Problem Domain Semantics

The refined conceptual model of the domain called for the identification of problem classes with attending semantics as a better means of capturing domain experience and commonly occurring patterns. As discussed in the survey of related research (Chapter 1), problem classes within the domain frequently serve as a means for discussing implementation techniques or programming language features. What is not present is a conscientious effort to describe, in-depth, the characteristics and concepts (i.e., semantics) of the problem classes and thus the entire domain.

An extensive study of production software was called for to identify the semantics. During this case study, commonality in programs had to be abstracted from variations in implementation techniques. The need for this abstraction process was predicted first by the study of the student programs, all of which solved a single problem, and later reinforced in a controlled experiment as reported in Chapter 2. Terminology had to be defined to describe concepts and commonly occurring variations observed during the
analysis of the programs. Then a mechanism for organizing the observations and terminology i.e., a way of representing the semantics, had to be found. Constructs of programming languages, such as Cobol and PL/I, have little understanding of problem domain semantics, but they have precise language syntax. Since representing domain semantics was of primary concern, a mechanism called templates was adopted. Templates are more descriptive of domain semantics than are programming languages.

Commonality discovered during the analysis phase of the case study was codified into the templates. Thus, the knowledge accumulated by experienced programmers can be used to guide and standardize activities of subsequent programming efforts through the application of a template-based methodology.

3.1 Case Study

3.1.1 General Data

Forty programs were collected from two major corporations to reflect a large variety of templates. The sample programs contained 76,103 lines of source code (later referred to as LOC). A statement in Cobol can span several lines and the number of lines of code was determined by using the internal line count (leftmost column) supplied by the compiler on the source listing. The average program contained 1,691 source lines, with 55% of these belonging to the
procedure division and most of the remaining 45% belonging to the data division.

Table 7 shows an estimated range of development times which might have been required to develop the 76,103 LOC in our sample. Lacking actual project development data, the objective is to use this table to show that the sample represents a significant amount of programmer experience. Table 7 only estimates development time for new code, but a programmer's time is split among development, maintenance, training, etc. Brooks cites one study in which it was found that programmers only spent 50% of the working week in actual programming and debugging time [Broo75]. Other studies show that a programmer spends over 80% of her/his time in maintenance activities [Boeh73]. Assuming that the only programmer activities are development and maintenance and using a conservative 50-50 allocation of time between these activities, a conservative estimate of the actual programmer time incurred in developing and maintaining the sample LOC might be twice the time incurred in developing new LOC.

The estimates in Table 7 were taken from research on samples with characteristics closely related to the case study sample [Wals77], [Fox82]. The productivity figures taken from Fox are averages of productivity figures reported for programs within each category. Using these estimates of productivity, a range varying from 11.2 years to 25.6 years might have been needed to develop the
programs in the sample. The formula used to compute these estimates is given in Table 7. If these figures are doubled to allow for maintenance activities on the sample LOC, as previously argued, the sample represents 22.4 to 51.2 years of programmer experience within the problem domain. Even the lower figure represents an impressive collection of experience.

Table 7: Estimates of Programmer Development Time Represented by the Case Study Sample

<table>
<thead>
<tr>
<th>Productivity Formula per man year</th>
<th>Sample Development Time in man years</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC/274/12 [Wals77]</td>
<td>23</td>
<td>Using medium productivity figure</td>
</tr>
<tr>
<td>LOC/566/12 [Fox82]</td>
<td>11.2</td>
<td>small/easy</td>
</tr>
<tr>
<td>LOC/280/12</td>
<td>22.6</td>
<td>medium/easy</td>
</tr>
<tr>
<td>LOC/248/12</td>
<td>25.6</td>
<td>medium/hard</td>
</tr>
</tbody>
</table>
3.1.2 Rationale for the Case Study

The objectives of the case study were to:

- Provide a reasonably complete semantic characterization of the file/database domain (that is, identify a template set that can support most programming tasks in that domain.)

- Establish that it is possible to define templates that strike a balance between strong semantic content and extremely limited applicability (that is, each template has to be as meaningful as possible and still support application specific variations).

- Establish that there is a need to combine templates in a variety of non-trivial ways, and to identify the approaches to be used.

- Identify the problems in providing semi-automatic, human-engineered support for the templates.

3.1.3 Techniques Used in the Case Study

The approach was to take an existing template set - the Hypothesis Set - and to enhance it using the case study (see Figure 6). The Hypothesis Set [Rama82] was defined based on the PDM Template Set [West79]. The existence of the PDM Template Set helped establish two points. First, there was a body of knowledge, in an identifiable domain, that could be represented as templates. Second, existing query-based systems did not support many software development efforts and, thus, template approaches were useful for more complex software efforts.

The case study data summary forms for each template included any newly discovered aspects (e.g., option, design/decision
alternatives) of each template in the Hypothesis Set. The template data, so collected, was used to justify the Enhanced Template Set. The summary data forms for each template with discussions and analysis information are included in Appendix C.

3.1.4 Hypothesis Template Set

The definition of a "more complete" Hypothesis Template Set was accomplished by adding enhancements to the PDM Template Set. The PDM Template Set, Hypothesis Template Set, Enhanced Template Set, and relationships among the three template sets is illustrated in Figure 6 and given in Table 8 along with a brief description.

It was also assumed that the Hypothesis Templates could be combined in a linear sequence using external files3. For example, if a problem required the Validate and the Report templates, then the Validate template would be applied first, resulting in an output file of records which, in turn, would be the input for the Report template.

3A solution inspired by the lack of a co-routine feature in the languages commonly used in this problem domain.
Figure 6: Refinements of Template Sets
Table 8: Comparison of The Three Template Sets

<table>
<thead>
<tr>
<th>PDM Templates</th>
<th>Hypothesis Templates</th>
<th>Enhanced Templates</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exclusive-Or</td>
<td>C</td>
<td>same</td>
</tr>
<tr>
<td></td>
<td>Series</td>
<td>L</td>
<td>same</td>
</tr>
<tr>
<td></td>
<td>Select</td>
<td>S</td>
<td>A more complex input structure with &quot;C* placed at a selected level</td>
</tr>
<tr>
<td></td>
<td>Series w/ Iteration</td>
<td>E</td>
<td>Similar to above comment except using &quot;L&quot;</td>
</tr>
<tr>
<td></td>
<td>Match Sequential</td>
<td>I</td>
<td>The &quot;I&quot; template can generate code for any set of inputs and allow for multiple sources, record types in either a fixed or arbitrary order, logical record groups defined several ways. Used to generate input traversals for all templates needing them. Unifies several specialized templates from the earlier sets.</td>
</tr>
<tr>
<td></td>
<td>Match Random</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail Report</td>
<td>Detail Report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary Report</td>
<td>Summary Report</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail/</td>
<td>Detail/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td>Summary</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td>Because of shared semantics the three templates were combined into &quot;X&quot;. The new template also contains numerous options, terms, and checklists.</td>
</tr>
</tbody>
</table>
Table 8, (continued)

<table>
<thead>
<tr>
<th>PDM Templates</th>
<th>Hypothesis Templates</th>
<th>Enhanced Templates</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update Master File</td>
<td>Update</td>
<td>U Additional variations and options included.</td>
<td></td>
</tr>
<tr>
<td>Edit Sort File</td>
<td>Validate</td>
<td>V A major revision of earlier templates with a large increase in semantics.</td>
<td></td>
</tr>
<tr>
<td>Edit Trans File</td>
<td>Sort</td>
<td>O same</td>
<td></td>
</tr>
<tr>
<td>Convert Data File</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort Table File</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract Table File</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T A new template which uses &quot;T&quot; to produce new files.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>H Iterate over a multilevel table for unspecified purposes.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A Initialize a table using fixed values.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B Initialize a table by placing records from a sequential file sequentially in the table.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>J Initialize a table by directly placing records into it.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F Indexed linear search of a table.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>G Linear search of a table.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P Binary search of a table.</td>
<td></td>
</tr>
</tbody>
</table>
3.1.4.1 Site and Program Selection

Two major corporations\(^4\) with large data processing organizations were selected as test sites. Each corporation supplied a representative sample of 20–25 programs from the problem domain. Guidelines and restrictions on the kind of software to be collected were kept to a minimum. The programs were to be file or database oriented. Because of the manual analysis to be employed in the case study, programs written using structured techniques were preferred. Programs written for transaction processing systems, such as CICS\(^5\), were not included in the study. Several complete systems with numerous programs were submitted by the companies. The rationale was, that by analyzing a complete system, the likelihood of obtaining dissimilar programs would increase. Because discovery of domain semantics was the goal, the participating corporations

\(^4\)PPG INDUSTRIES INC. a major manufacturing corporation which produces automotive glass and paints as well as glass and paints for other uses.

MELLON BANK INC. one of the largest corporate and consumer banks in the country which provides data processing services for numerous other financial institutions.

\(^5\)A communications oriented IBM program product where the user interface is in terms of transactions which are identified by codes. The system software knows how and when to invoke processes for each transaction type. The individual processes are much more limited in function with the system being responsible for invoking the correct process sequences as defined by the system programmer.
were asked to supply programs of as many different types as could be identified, instead of taking a random sample which might yield too many programs containing the same template type.

3.1.4.2 Program Analysis Techniques Used

After the logging of each program, the first step during program analysis was to gain an overview of the program logic. Application details were not necessary to achieve this, although it frequently aided the analysis. Documentation overviews (collected with the programs) frequently described programs using terminology present in the Hypothesis Template Set, although, without a detailed analysis, the commonality among programs described by identical terms in the documentation was not readily apparent.

There were many problems in identifying whether the hypothesized templates occurred in the collected programs. These were due to the following:

- There was a large variety of implementation techniques used.
- A variety of manual optimization techniques were used by the programmers to optimize the number of reads and writes to secondary storage.
- There were major requirement variations within a given template.

The central challenge in defining a template is to separate superficial differences, such as implementation style, from major variations inherent in the problem class described by the template.
To meet this challenge, the requirements, design and implementation of each program had to be thoroughly understood (at least to the degree needed by a maintenance programmer). To this end, three forms were completed for each program. The first documented the existing program. The second applied templates to the problem solved by the original program, and the third recorded demographic data. Each form with descriptions of their contents is given in Appendix G.

After the three program study forms were completed for each program, the study shifted from a program-centered emphasis to an emphasis on the use of individual templates across programs. A data collection form was designed from the definition of each template in the Hypothesis Set (see Appendix C). A representative sample of the programs that used the semantics of an individual template were then used to complete the data collection form for that template. The sample was chosen to show a variety of semantics which was consistent with the goal of the study. This process was then repeated for each template in the Hypothesis Template Set. Additional templates were defined to describe commonly occurring patterns found in the program sample, but not covered by the Hypothesis Template Set. These additional templates were also used to define data collection forms which, in turn, were used to record case study findings dealing with that template. The average time required to complete the forms for each program and the templates in
the program was approximately 12 hours.

During both the program-centered analysis and the template-centered data collection phases, concepts, variations, options, and terminology descriptive of each template were synthesized. The results of this synthesis are reported in a detailed discussion of the semantics of each template in Appendix C and in a summary form in Appendix D. An understanding of the discussion accompanying each template is needed to correctly interpret the case study data and the summary template definition. The following sections describe some of the results of the case study in more detail.

3.2 Semantic Characterization of the File/Database Domain

Jensen and Tonies, in their text on software engineering [Jens79], discussed the need, during system design, for understanding the semantics of the problem and defining terms to fit those semantics. They stated, "we cannot understand the system unless we can find the proper words to describe the system."

"Finding appropriate terminology is the heart of the design process itself." This author would agree with their assessment and hasten to add that a more difficult problem might be in deciding on the concepts that are critical to the problem domain and that must be described by the terminology. One major contribution of this study is to identify these concepts and provide appropriate terminology and the meaning of that terminology.
In the next subsection a presentation of basic concepts will appear and will be used to discuss why programs are difficult to write in the file/database processing domain.

3.2.1 Concepts Used in the Characterization of the Domain

Typical problems in the file/data base domain require programs to:

1. traverse fairly intricate and complex networks of records (or nodes) in secondary storage,

2. create, in program memory, a group of records, meeting certain conditions, which need to be "processed" together,

3. process the records, and

4. create groups of processed records to be written out.

Preliminary studies [TRIA80] undertaken by the TRIAD research group at Ohio State University show the following:

- The typical file/database problems are complex because of the intricate control structures needed to perform tasks 1, 2 and 4 described above.

- The code written to process each group of records in memory is fairly simple.

- The task of locating the processing code correctly within the complex control structure is difficult.

Automatic program generation systems and query systems have been developed (e.g., SQL, FOCUS) for the domain. However, their use has been limited due to the inability to traverse complex structures and manipulate data in a variety of flexible ways. The
Westinghouse PDM effort, which was chosen as a starting point for defining the Hypothesis Template Set for use in the case study, is one of the more sophisticated efforts at templates for supporting complex traversals as well as allowing for flexible data manipulations. This effort established that the domain was indeed amenable to a semantic characterization.

Expert programmers in the file/data base domain have identified classes of problems and, for each class, a generic solution (e.g., the Westinghouse PDM effort.) The generic solutions have meaningful names and provide a semantic characterization of the domain. They typically provide a control structure for traversing the files/databases and grouping together the relevant records for further processing. In the case study, additional support for obtaining solutions to classes of problems by enhancing the templates and template set with semantics (in the form of prompts, help, check lists, etc.) was identified.

Basically, a template definition is an attempt at precisely identifying the various words that would support the problem solving task. In Appendix C these words are defined for each template. The empirical evidence from sample programs is obviously interpreted subjectively. Thus, the choice and definition of terms are also subjective.
3.2.2 Impact of the Case Study

The major contribution of the case study was in identification and definition of semantics for the problem domain and organization of these semantics into individual templates. The case study also provides valuable insight into the need to combine templates in a flexible manner, the nature of this combination or interfacing of templates, the I/O efficiency of template combinations, and the need for a powerful, user-friendly interface to lead the user through the complex semantics of templates and template combinations. Each of these contributions will be discussed in the following sections.

3.2.2.1 Addition of Semantics to Individual Templates

The following list summarizes how individual templates in the Hypothesis Set were enhanced to get the final templates described briefly in Table 9.

- Addition of more semantics, that is, precisely defined terms. For example, the definition of trigger, disposition, validation, report and process test types were added to the Validate template.

- Addition of new templates, namely, the Fabricate and Table templates. The Hypothesis Template Set was file oriented and did not process internal data structures (tables). The case study showed the importance of table templates to be used in combination with other templates and to be included in the definition of more complex templates such as Validate. A hierarchy of table oriented templates was therefore defined. At least 18 programs contained the Fabricate template and at least 31 programs contained one or more Table templates.

- Addition of file/database traversals that allow for
multiple sources, multiple record types and multiple logical groupings of records. The logical groupings can be defined by position, key fields or delimiters. The **Generalized Traversal** template now allows for all of these in the code for input traversal.

- Inclusion of commonly occurring specification variations in each template. For example, **Update** allows for fixed master-identity as well as enumerated master-identity updates.

- Presentation of choices. When the user reached a certain point, the template could present possible paths to follow. For example, there were several methods of reporting errors within **Validate** and there were many methods possible for disposing of tested records.

- Incorporation of help displays in each template for educating the inexperienced user or reminding the expert of the information needed by the template.

To provide insight into the nature of the semantics that might be added to enhance a template set (and push the coverage as illustrated graphically in Figure 6), Appendix E compares the semantics of the PDM Template Set using the semantics of the Enhanced Template Set and the semantics of the Hypothesis Template Set using the semantics of the Enhanced Template Set.

### 3.2.2.2 Completeness of the Enhanced Template Set

The completeness of a template set can be measured by attempting to generate programs in the sample. However, without an operational template-based environment, it is difficult to accurately gauge the number of programs for which documentation and skeletal code could be automatically generated by the Enhanced Template Set. Another difficulty, in measuring the percentage of
<table>
<thead>
<tr>
<th>Template Name</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Template to access and process an arbitrarily complex set of data structures.</td>
</tr>
<tr>
<td>C</td>
<td>Template which provides a mutually exclusive set of process locations whose execution is controlled by an ordered set of conditions.</td>
</tr>
<tr>
<td>L</td>
<td>Template which provides an ordered sequence of process locations whose execution is controlled by an independent set of conditions.</td>
</tr>
<tr>
<td>H</td>
<td>Template to traverse an arbitrarily complex table structure. It provides process locations for unspecified processes.</td>
</tr>
<tr>
<td>A</td>
<td>Template to initialize/reinitialize a table using a fixed value(s).</td>
</tr>
<tr>
<td>B</td>
<td>Template to initialize/reinitialize a table by sequentially placing records from a sequential file.</td>
</tr>
<tr>
<td>J</td>
<td>Template to initialize a table using records from a sequential file where the records are placed directly (i.e., under the control of a subscript contained in the record) into the table.</td>
</tr>
<tr>
<td>F</td>
<td>Template to perform a binary search on an indexed table using the Cobol &quot;SEARCH ALL&quot; verb.</td>
</tr>
<tr>
<td>F</td>
<td>Template to perform a linear search on an indexed table using the Cobol &quot;SEARCH&quot; verb looking for an entry(ies) which meet a specific condition.</td>
</tr>
<tr>
<td>G</td>
<td>Template to access every entry in a table searching for an entry(ies) which meet a specific condition.</td>
</tr>
<tr>
<td>S</td>
<td>Template which repeatedly executes the C template.</td>
</tr>
<tr>
<td>E</td>
<td>Template which repeatedly executes the L template.</td>
</tr>
<tr>
<td>O</td>
<td>Template which accepts unordered records in a file, orders these records by specified key fields, and places the ordered records in an output file. The template uses the Cobol &quot;SORT&quot; verb.</td>
</tr>
<tr>
<td>U</td>
<td>Template which applies transactions to an existing information model called the master in order to bring the master up to date.</td>
</tr>
<tr>
<td>V</td>
<td>Template used to determine the correctness of records as defined by a set of test criteria.</td>
</tr>
<tr>
<td>X</td>
<td>Template which traverses 1 or more sources of data, provides process code location so that output can be formed from selected input records in accordance with relationships among the sources. Further, this template includes semantics which deal with the format of the output report.</td>
</tr>
<tr>
<td>Y</td>
<td>Template which manipulates 1 or more sources of data to produce output data in a new form, format or organized according to new criteria.</td>
</tr>
</tbody>
</table>
code that could be automatically generated, follows from the fact that a template-based methodology would probably lead to solutions that differ from programs generated without the benefit of such a methodology.

An attempt was made to determine the completeness of the Enhanced Template Set by discerning individual template use in the actual case study programs. Table 10 summarizes this data. When accumulating the counts, an attempt was made to identify, within the program, the template with the most semantic content and record its use. For example, Validate can generate detail and/or summary reports. In the data summary, only Validate was counted, not Validate and Report. The Report template would be counted only when it was not part of another template. A curious side effect of this decision is that the Generalized Traversal template, whose semantics are present in most of the other templates and which is therefore heavily used, is seldom used independently and thus is not counted very often.

The Enhanced Template Set was found to be fairly complete. Though new templates were added -- Fabricate and the Table templates -- their semantics were already "embedded" to some extent in the Hypothesis Template Set. The decision to define separate templates was made because they were recognizable patterns that were either frequently used separately (as was the case with Fabricate), or their semantics were embedded as a group in the definitions of other
Table 10: Template Usage in the Program Sample

<table>
<thead>
<tr>
<th>Enhanced Set Templates</th>
<th>Count of Programs Containing Template</th>
<th>Total Count of Template Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>G</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>H</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>O</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>U</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>V</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>X</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>Y</td>
<td>18</td>
<td>27</td>
</tr>
</tbody>
</table>
templates (as was the case with the Table templates). What the case study did achieve was to add a considerable amount of semantics to the individual templates.
Chapter 4

Description and Discussion of Semantics for a Representative Template

The case study resulted in a large volume of material being generated for each template. This chapter serves to present the types of results obtained for a representative template. The first section summarizes the Validate Template semantics and is useful in gaining an overview of the template both in terms of the semantics included in the template and the relationship of those semantics with other templates in the set. The second section contains a detailed discussion and definition of the Validate semantics and how to interpret the data summary tables that record the case study findings for that template. The third section is the actual data summary tables resulting from the recording of case study findings for Validate. The final section uses the data summary categories developed during the case study to rate appropriate (those dealing with validate-like activities) PDM templates and the appropriate Hypothesis Set template.
4.1 Summary of the Validate - V Template

This is the definition of the Validate template which was designed based on data collected during the case study. The actual data collected for Validate is presented in the Sections 4.3 and 4.4.

4.1.1 Related Templates

Validate is used to determine the correctness of records as defined by a set of test criteria. The definition of Validate is a synthesis of other templates - I, X, H, A, B, J, F, G, P, C, L where:

- I is used to provide the control for the traversal of subject and object sources to be tested.
- X is used to generate several possible types of reports.
- A, B, J are used to initialize tables which play a prominent role in supplying test objects, in the setting and testing of summary validity flags, and in generating error messages.
- H, F, G, P are used to access table elements.
- C, L are used to construct the test types - validation, process, disposition, or report.
4.1.2 Terminology

Based on an analysis of the template semantics, the template terminology and concepts can be classed as dependent or independent, and inessential or essential. Dependent semantics "belong" to the template and are not borrowed by embedding the semantics of another template. Independent semantics on the other hand have been borrowed from another template. Essential semantics must always appear, in some form, in all problems for which the template is applicable. Semantics which may or may not be present are called inessential. What follows is a summary of the Validate template semantics. A more detailed discussion is provided in the next section.

Terminology describing the variations in the specifications that are essential for validate problems is given next.

1. **Source of records** to be validated - the fields of these records will become the subjects of test triggers and validity tests.

2. **Disposition of tested records** - once the validity or invalidity of a record is determined, the record must be disposed of appropriately.

3. **Test Types** - five test types are defined, but only the validation test is essential. The validation test consists of a test "trigger" or condition and a validity test. The test trigger controls execution of the validity test. The validity test may apply several criteria (by means of tests) to determine the correctness of a record. Other test types are process tests, disposition tests, report tests, and group trigger tests. These are all dependent but optional, thus inessential.
4. **Test triggers** present in validation tests, group test triggers, and disposition tests can be one of the following:

- record type defined by record field value,
- record type defined by position in the structure,
- structure type,
- previous validation test results,
- null, i.e., always true,
- combination of test trigger types,
- record field values other than record type,
- group validation test results, and
- structure validity.

5. **Validity test types** are as follows:

- between fields of a single record,
- field completeness,
- class test,
- range test,
- set membership implemented by one of seven different methods,
- fixed values,
- duplication,
- group structure,
- combination of above.

6. **Validity Test Subjects and Objects** not provided by the *(Generalized Traversal)* Template, i.e., local to a specific test include: fixed value, computed value,
access an internal structure, and access an external structure.

7. **Recording of Validation Results** - the results of the testing process for each record or possibly group of records, (i.e., structure) is recorded by setting flags, partitioning of output, and/or report generation.

8. **Disposition of tested records** is accomplished by one or a combination of the following:

- Setting of a flag,
- Outputting both valid and invalid records,
- No partitioning used,
- Partitioning valid from invalid,
- Partitioning within invalid,
- Output neither valid nor invalid,
- Use valid immediately (in synchronization with another program),
- Correct invalid records,
- Output valid only,
- No valid record partitioning,
- Valid record partitioning,
- Output invalid only,
- No invalid record partitioning,
- Invalid record partitioning.

The following terminology is unique (dependent) to the **Validate** problem class but is not necessarily present in every validate problem (i.e., inessential):
1. **External source(s) of test objects.**

2. **Internal source(s) of test objects.**

3. **Report generation** which could be one or a combination of several of the following:
   - reporting deferred to report test,
   - printing valid record,
   - printing invalid record with error message,
   - printing invalid record with error flag,
   - printing error message only.

4.2 Discussion of Validate Semantics

**WHEN IS THE TEMPLATE APPLICABLE?**

The **Validate** template is applicable when a source of data, grouped into records or nodes, is of questionable quality. Confidence in the correctness of the data is to be increased by subjecting the data to a number of tests. The results of these tests need to be communicated. Also, the results are used to determine the disposition (i.e., the storage or processing) of the tested records.
4.2.1 Template Semantics and Related Case Study Observations

At a high level, the major components of the Validate template are:

1. A source of records of questionable quality.
2. A set of tests to determine the correctness of the records.
3. A means of recording of the results of the testing.
4. A means of disposing of the tested records.

Each of these components is discussed below.

4.2.1.1 Record Sources and Destinations

Many varieties of both external data structures (files, databases) and internal data structures (tables, arrays, stacks, etc.) are used to either supply the record (which may be a record, a segment, a tuple, or a structure) to be tested (hereafter, called the subject) or to supply the item to be tested against (hereafter, called the object). Each input or output structure is identified by its purpose, name, access method needed (that is sequential, relative, random, or dynamic), any relationship(s) existing among records of the structure, and the format of each unit to be accessed. Therefore, the following categories are used to describe record sources and destinations:

1. Input source of records to be validated (subjects of tests).
2. External source(s) of records to be used to test against (i.e., objects of tests).

3. Internal source(s) of records to be used to test against (i.e., objects of tests).

4. Reports to be generated.

5. External destinations of tested records.

6. Internal destinations of tested records.

4.2.1.2 Tests

**TEST TYPES**

- **Validation tests** that compare a subject with an object are the most common test type. A validation test consists of a test trigger containing a condition that determines when the validity condition is applied. When several validity tests have the same trigger, a group validation trigger may be defined to control the execution. The group trigger is a form of manual optimization that could be done by automatic means.

- **Process tests** that control a process to be applied selectively without regard to any specific validation test.

- **Disposition tests** used to decide what to do with a record if the disposition action is not controlled by an individual validation test. Disposition tests are used when a group of validation test results determine record disposition.

- **Report tests** are used if the generation of report lines (with subject records and/or error messages) is not controlled by the individual validation test.
TEST TRIGGERS

Triggers for use in validation tests or disposition tests, or as group validation triggers may be provided in a variety of ways as shown by the following:

1. Record type defined by a field within a record.
2. Record type defined by the position of the record within a structure.
4. Previous validation test results.
5. Absence of any trigger at the individual test or group level, which indicates that the test is to be performed on all the records.
6. Combination of trigger types.
7. Values of certain fields of each record other than those defining record type.
8. A group of validation test results.
10. Presence of a group trigger (but no individual trigger).

Several of these categories are closely related and, at times, a subjective judgment must be made as to which of the categories should apply. For example, differentiating between record type (trigger type 1) and record field values (trigger type 7) depends on understanding the specific application because, at the syntactic level, they are frequently identical. Trigger type 4 relies on "previous test results" either by an explicit test of an error flag or by a group of tests (using a case structure) that are only
evaluated until one of the tests is true. So a test in the case structure is triggered only if all the previous tests are false.

Validity tests are frequently grouped together and controlled by several separate triggers. This is a form of manual optimization that requires the code for a validity test be written only once but that triggers, remotely set, be used to invoke the validity tests. This optimization often leads to additional triggers preceding specific validity tests which were grouped with several other tests. Consider a group of validity tests, say \( V_1, V_2, V_3, V_4 \) which are all executed whenever triggers \( t_1 \) or \( t_2 \) or \( t_3 \) are true. A manual optimization would group the three triggers into a single compound condition.

\[
\text{IF } t_1 \text{ OR } t_2 \text{ OR } t_3 \text{ \ PERFORM TESTS-1-4-PARAGRAPH}
\]

This is an example of a group trigger controlling several validity tests. The need to place additional triggers on an individual test, say \( V_2 \), arises when the programmer discovers an additional condition say \( t_4 \) which, when true, causes \( V_1, V_3, V_4 \) to be executed but not \( V_2 \). The addition of triggers or the repetition of portions of group triggers, at best, is confusing and often leads to inefficiencies (i.e., retesting of triggers).
Several categories of validity tests were observed and are described in Table 11.

**Table 11: Categories of Validity Tests**

<table>
<thead>
<tr>
<th>Category Number</th>
<th>Category Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Between fields of a single record</td>
</tr>
<tr>
<td>20</td>
<td>Field completeness</td>
</tr>
<tr>
<td>30</td>
<td>Class test</td>
</tr>
<tr>
<td>40</td>
<td>Range test</td>
</tr>
<tr>
<td>50</td>
<td>Set membership</td>
</tr>
<tr>
<td>51</td>
<td>Compound condition</td>
</tr>
<tr>
<td>52</td>
<td>Case structure</td>
</tr>
<tr>
<td>53</td>
<td>Linear search of a table</td>
</tr>
<tr>
<td>54</td>
<td>Random access to a table</td>
</tr>
<tr>
<td>55</td>
<td>Random access of an external structure</td>
</tr>
<tr>
<td>56</td>
<td>Sequential access of an external structure</td>
</tr>
<tr>
<td>57</td>
<td>Use of condition names</td>
</tr>
<tr>
<td>60</td>
<td>Fixed value test</td>
</tr>
<tr>
<td>70</td>
<td>Duplicatiion</td>
</tr>
<tr>
<td>80</td>
<td>Group structure</td>
</tr>
<tr>
<td>90</td>
<td>Combination of several of the above</td>
</tr>
</tbody>
</table>

If both the subject and the object of a test come from the source of records to be validated, then validity test category 10 is used. The "Between Fields of a Record" test attempts to establish
the correctness of a relationship between fields within a single record; for example, employee age and length of time since last physical exam, where both the age and date of last physical are fields in the employee record.

"Field Completeness" tests attempt to establish that fields of a record do contain values other than default or initialization values. The predominant use of this test compares fields with spaces.

"Class" tests compare field values to ensure the value is of the correct type, i.e., numeric, alphabetic, or some combination of these two.

"Range" tests are normally used with numeric fields to determine whether the value is within an allowable range (e.g., hours worked greater than 0 and less than 60). Range tests could also be used to test alphanumerical fields relying on the collating sequence to determine ordering within the range. A predetermined ordering must be in effect to utilize this validity test type.

"Set Membership" tests differ from "Range" tests because allowable values are enumerated in set membership tests whereas allowable values are defined by bracketing values with range tests. It is assumed that the set contains two or more elements. This differentiates this category from a type 60 (a "Fixed Value") test. A variety of techniques are employed in implementing "Set Membership" validity tests. Data was collected on each of the
implementation techniques used. Individual programs frequently employed several of these techniques which leads one to speculate on why one technique was selected over another. Possible answers are:

1. A table or file had already been defined for the set in question.

2. The number of elements in the set was large and this made compound conditions, case structures or 88 levels inappropriate.

3. Management standards specified the kind of implementation to be used.

4. A very large number of elements might have suggested an external structure, thus conserving program memory.

The "Fixed Value" test (type 60) is similar to "Field Completeness" (type 20) when the fixed value is a default or a initialization value. "Field Completeness" restricts test objects to default values such as spaces or zeroes, which accounts for the distinction between "Field Completeness" and "Fixed Value" categories.

"Duplication" is included for completeness, although none of the programs studied executed a "Duplication" test. A duplication test determines if two or more records with matching criteria exist in the source structure.

The "Group Structure" tests are the most complex because the program must deal with tests that bridge more than one record. This requirement necessitates remembering data from previous records, determining the correct time to invoke the test, and updating the
memory of the structure. Program D170 from the case study sample is an excellent example of how complex this type of test can become. In this program, for a record to be valid, it must have a "support" record of a specific type with a key value equal to the key value of the record in question. The file being validated is first checked for the support record. Should none exist, a master file is then checked for the support record.

During the case study, compound validity tests, which occurred as compound conditions were treated as separate tests where possible. If it was not possible to separate a compound test into its component tests it was classified as a type 90 "Combination". The following condition is an example of a "Combination" test made up of types 30 and 20 which was not separable into two tests. Here a remote group trigger is assumed for the sake of simplicity.

```
IF HOURS NOT NUMERIC AND
   HOURS NOT = SPACES
<error processing>
```

The test recognizes a blank value or a numeric value as valid. An attempt to separate the compound test into two individual tests causes problems because either it is not logically equivalent or leads to two equally complex tests.
(Not Logically Equivalent)
  IF HOURS NOT NUMERIC
  <error processing>

  IF HOURS NOT = SPACES
  <error processing>

An example using nested tests and one that is logically equivalent to the above compound condition, but leads to a more complex structure is:

(Logically Equivalent, But Equally Complex)
  IF HOURS NOT NUMERIC
    IF HOURS NOT SPACES
    <error processing>

That is, it is not possible to separate the two conditions in the above compound condition into two distinct, logically equivalent validity tests. Hence, the compound tests were treated as a "Combination" test.

The following compound validity test combines two separate validation tests with common triggers and identical valid (no action) and invalid (<error processing>) actions into a single test. This kind of compound test is recorded as two separate validity tests.

  IF DEPT = 60 OR RETURNED-GOODS = "NO"
  <error processing>
In terms of the tests performed it is equivalent to:

IF DEPT = 60

<error processing>

IF RETURNED-GOODS = 'NO'

<error processing>

Whenever possible, the compound tests were broken into individual tests in order to obtain a more accurate description of the testing semantics.

VALIDATION TEST ACCESS

A validation test is defined as a test trigger followed by the validity condition which tests a subject against an object. The discussion here concerns the use of the Generalized Traversal template to ensure that the proper subject and object are present in memory before the validation test is executed.

In the case study, the test subject was usually available before execution reached the test because of a global traversal of the source(s). Therefore, there was usually no need for accessing the subject source as part of the individual test, that is local access was not needed for the subject of the test. An example where local access of the subject is needed would be a 1 record look-ahead to guarantee that the correct record followed the current record. A second example of local subject access would be the existence of a
table or a repeating group within the record being tested.

An object can be one of four types:

1. A fixed value (constant, parameter or class),
2. A computed value,
3. Accessed from an internal data structure,

Types 1 and 2 assumed the value is in memory and is identified by a constant or a variable name. Types 3 and 4 required a traversal of a structure to produce the object(s) to be used in the test. This traversal could be local to the test or could be global, i.e., used by several tests. When a local traversal is needed, a template to access the structure needs to be embedded.

4.2.1.3 Reporting

Validation often includes the generation of a detailed and/or summary report which is printed by the program doing the validation, or placed in a file that is accessed on-line, or placed in a file to be printed at a later time by another program. Reporting can be done at the time an individual validation test is performed or set up at that time by formatting the needed lines and deferring their output to a report test. Error messages can be given for each error discovered or a summary message containing all the errors can be printed. The rich variety of possible choices led to the reporting categories listed in Table 12. Combinations of reporting categories
Table 12: Reporting Categories

<table>
<thead>
<tr>
<th>Reporting Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No reporting at the individual test level, defer to report test.</td>
</tr>
<tr>
<td>1</td>
<td>Print valid records.</td>
</tr>
<tr>
<td>2</td>
<td>Print invalid record followed (on separate line) by the error message.</td>
</tr>
<tr>
<td>3</td>
<td>Print invalid record preceded (on a separate line) by the error message.</td>
</tr>
<tr>
<td>4</td>
<td>Print invalid record with error message (on the same line).</td>
</tr>
<tr>
<td>5</td>
<td>Print invalid record with error flag (on same line).</td>
</tr>
<tr>
<td>6</td>
<td>Print error message only.</td>
</tr>
</tbody>
</table>

are possible.
4.2.1.4 Disposition of Records After Completion of Testing

When validation tests for a record are completed, prior to accessing the next record to be processed, disposition of the current record must occur. This disposition can be controlled by the validity of the record and/or the validity of the structure containing the record. Disposition may or may not include flagging the record and may or may not include partitioning records by validity and/or record type. Table 13 contains a list of disposition actions.

<table>
<thead>
<tr>
<th>Disposition Category</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flagged or Not Flagged</td>
</tr>
<tr>
<td>10</td>
<td>Write both valid and invalid</td>
</tr>
<tr>
<td>11</td>
<td>No partitioning</td>
</tr>
<tr>
<td>12</td>
<td>Partition Valid from Invalid</td>
</tr>
<tr>
<td>13</td>
<td>Partition within the Valid</td>
</tr>
<tr>
<td>14</td>
<td>Partition within the Invalid</td>
</tr>
<tr>
<td>20</td>
<td>Write Neither</td>
</tr>
<tr>
<td>21</td>
<td>Use Valid immediately</td>
</tr>
<tr>
<td>22</td>
<td>Fix Invalid Using heuristics</td>
</tr>
<tr>
<td>30</td>
<td>Write Valid Only</td>
</tr>
<tr>
<td>31</td>
<td>No partitioning</td>
</tr>
<tr>
<td>32</td>
<td>Partitioning</td>
</tr>
<tr>
<td>40</td>
<td>Write Invalid Only</td>
</tr>
<tr>
<td>41</td>
<td>No Partitioning</td>
</tr>
<tr>
<td>42</td>
<td>Partitioning</td>
</tr>
</tbody>
</table>

While categories 10, 20, 30, 40 appear to be mutually
exclusive, it should be noted that combinations of categories are possible. One program produced multiple disposition files - one with no partitioning, one with valid records of a specific type and one with invalid records of a specific type. This was recorded as 10, 11, 12, 13, 14.
4.3 Overview of Programs Using the Validate Template

The programs containing validate template semantics used in accumulating the study totals are listed next with a brief comment about each program.

1. BUCMSC06 -- A PL/1 program which nested validate within an update of an IMS database. Because of the nesting, valid records were used at once and not output to an external structure. Transactions to be validated were either in a one-record or two-record group. In the case of the two-record group, a look-ahead access was used.

2. AIT100 -- Noteworthy because of its use of a random access file and a relative file as sources of objects for validation tests. Records to be tested were grouped by header records with no grouping field on the individual record. Because record validity depended on group validity determined by computed totals, individual records were written to a temporary file until the group was completed. If the group was then determined to be valid, the records on the temporary file were written to a permanent file. This was done after each group.

3. CCP17 -- Records to be validated occurred in one of four possible types of group structure identified by a header record. Record disposition was determined by group validity tests but did not have to be delayed until the group was completed. Records could therefore be disposed of as soon as testing was completed for the individual record.

4. SDL010A1 -- Records were independent of each other. Tests relied heavily on internal tables that were searched linearly.

5. D100 -- Records were independent of each other. Tests relied heavily on internal tables searched linearly. This was by far the most massive program studied. Record type triggers, group triggers, and reuse of tests by using multiple remote triggers were common. Also, the program used a massive error summary flag that was set and passed on with both valid and invalid records.
6. D170 — This program determined the validity of structures in which the records resided. It was necessary for records to have valid predecessors with matching keys and be of the proper type. A check for the predecessor was first made in the file to be validated. If none was found, a matching of master file records with the record being validated was executed. A massive error summary flag was set and passed on with both valid and invalid records.
4.4 Summary Data Collected for Programs Containing the Validate Template

The terminology on the left was identified and defined during the case study. The columns on the right indicate whether this terminology was supported by the programs in the case study. Table 14 explains the notation used to report the summary data.

For a detailed discussion of the semantics refer to the previous sections titled, "Summary of the Validate - V Template" and "Discussion of Validate Semantics."
Table 14: Legend for Interpreting Notation Used in Reporting Summary Data

ID - "identification."

M - Many of the specific semantic category are allowed by the template. In effect, no limitations on numbers.

max - "maximum."

N - "no" indicating that no provision has been made for the semantic category.

N/A - the semantics do "not apply."

NR - "number" used for counting.

ran - "random" access.

rec - "record."

seq - "sequential" access.

x|y - identifies two distinct entries for a single program. For example, S1|S2 indicates two sources, S1 and S2, are being reported. Other entries for a program with two sources will use the "|" to indicate which of the two sources the data belongs to. For example, "Y|" indicates "Yes" for the first source. Similarly, " |N" indicates a negative response for the second source.

x*y - answer "y" is repeated "x" times.

x-y - Where x and y are integers, shows a range of answers in the semantic category. This would reflect flexibility in the template. By the same token, a single digit means the template can only deal with that specific variation and is thus limited.

x/y - Shows the two possible answers.

Y - "yes, indicates the semantics are present."
### A. Record Sources and Destinations

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

1. **Subject Sources**

<table>
<thead>
<tr>
<th>Number</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Seq</td>
<td>Seq</td>
<td>Seq</td>
<td>Seq</td>
<td>Seq</td>
<td>Seq</td>
</tr>
<tr>
<td>Rec Types</td>
<td>$2^6$</td>
<td>$2^7$</td>
<td>58</td>
<td>$2^9$</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>Group Levels</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

2. **External Object Sources**

<table>
<thead>
<tr>
<th>Number</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Ran</td>
<td>Ran</td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rec Types</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group Levels</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. **Internal Object Sources**

<table>
<thead>
<tr>
<th>Number</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>3</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Template</td>
<td>$3^F$</td>
<td>$4^F$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Levels</td>
<td>$3^1$</td>
<td>$4^1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

6. One or two card groups are present with the first card indicating if a second card exists. The cards also contain repeating groups in the form of a table.

7. Record groups are separated by group leader.

8. Four record group types; each group identified by a group leader.

9. No inter-record relationship.

10. A seven level hierarchy consisting of seven record types to be processed and several other types to be ignored.

11. Valid records used at once by nested UPDATE.
### 4. Reports

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Number</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Error only</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid only</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combination</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5. Disposition Files

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporary</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid only</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid only</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid-invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
B. Test

<table>
<thead>
<tr>
<th>TEST TYPE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Validation</td>
<td>16</td>
<td>14</td>
<td>3</td>
<td>7</td>
<td>249</td>
<td>9</td>
</tr>
<tr>
<td>2. Process</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>3. Disposition</td>
<td>1</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Report</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>54</td>
<td>2</td>
</tr>
<tr>
<td>5. Group</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TEST USE\(^{12}\)  

| TEST USE\(^{12}\) | 16 | 33 | 3 | 7 | 760 | 18 |

AVE TEST USE\(^{13}\)  

| AVE TEST USE\(^{13}\) | 1.0 | 2.4 | 1.0 | 1.0 | 3.1 | 2.0 |

\(^{12}\) Test use is a measure of the number of times a trigger is changed and the test used for a single record. It is computed by multiplying the number of different triggers used to control a validity test by the number of validity tests within the scope of each trigger. There was usually one trigger per validity test, but frequently, several validity tests would be grouped into a paragraph and executed under the control of different test triggers.

\(^{13}\) Test Use metric/number of validity tests in the program.
### TRIGGER TYPE

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Record Type</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By Field</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>93</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>2. Record Type</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>By Position</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. Structure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. Previous Test</strong></td>
<td>4</td>
<td>4</td>
<td>58</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>5. All Recs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td><strong>6. Combination</strong></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>7. Field Values</strong></td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>8. Disposition By</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Group Of Tests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td><strong>9. Disposition By</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>10. No Individual</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>14</td>
<td>13</td>
<td>2</td>
<td>7</td>
<td>151</td>
<td>1</td>
</tr>
</tbody>
</table>

### VALIDITY CONDITION TYPE

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10 Fields of a</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>20 Completeness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>30 Class</strong></td>
<td>13</td>
<td></td>
<td>1</td>
<td>115</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>40 Range</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>50 Set Membership</strong></td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>35</td>
<td>1</td>
</tr>
<tr>
<td><strong>51 Compound Cond</strong></td>
<td>5</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>52 Case</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>53 Table Linear</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>54 Table Random</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>55 File Random</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>56 File Linear</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td><strong>57 88 Levels</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>60 Value Test</strong></td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>113</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>70 Duplicates</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>80 Group</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td><strong>90 Compound</strong></td>
<td>11</td>
<td></td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### D. VALIDATION TEST SUBJECT ACCESS PATTERN

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Global-Pattern</td>
<td>I</td>
<td>3*I</td>
<td>I</td>
<td>259*I</td>
<td>9*I</td>
<td></td>
</tr>
<tr>
<td>2. External Source Local Access Template</td>
<td></td>
<td></td>
<td>2*R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Within Rec Local Access Template</td>
<td></td>
<td></td>
<td>2*I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### E. VALIDATION TEST OBJECT ORIGIN

<table>
<thead>
<tr>
<th>Program Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>1. Fixed Value</td>
</tr>
<tr>
<td>2. Computed</td>
</tr>
<tr>
<td>3. In-Core Structure Template</td>
</tr>
<tr>
<td>4. External Structure</td>
</tr>
</tbody>
</table>

14 Uses a 1 rec. lookahead read.
F. REPORTING

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INVALID</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Flagged</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Set Local</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Local Print Msg</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Local Print Rec</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Delayed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Msg Setting</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Msg Print</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rec Print</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Summary Msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td><strong>VALID</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Local Rec Print</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Delayed Rec Print</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. No Print</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### G. INVALID RECORD DISPOSITION DETERMINATION

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Disposition Test</strong></td>
<td>Y*3</td>
<td>Y*2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td><strong>2. Validation Test</strong></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. Structure Validation Test</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

### H. DISPOSITION ACTION

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Set Flag</strong></td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td><strong>2. Write Both</strong></td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Partition</td>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition V/IV</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition V</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition IV</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. Write Neither</strong></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Valid</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct Invalid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. Write Valid Only</strong></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Partition</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>5. Write Invalid Only</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Partition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.5 Comparison of Semantics

The main value of the case study has been in enhancing the semantics of the individual templates in the Hypothesis Set. This section contains the data summary tables developed during the case study for the validate problem class used to record the semantic content of appropriate (those dealing with validate problems) PDM and Hypothesis Set templates. This illustrates the limited semantics present in the predecessors of the Enhanced Template Set. The full details of all the PDM and Hypothesis Set templates compared in a similar fashion are available in Appendix E. The notation used to report results is explained in Table 14.

Examples of limitations in semantic content identified by this comparison are:

- Item A.2 "External Object Sources." The Edit Sort File PDM template only allows for sequential access. The Edit Transaction File template does not allow for an external source of objects at all.

- Item A.4 "Reports." The total number of reports allowed in any one of the three templates is only one and it is of type "Error Only."

- Item A.5 "Disposition Files." The two PDM templates only allow a single disposition file that must contain only valid transactions. The Hypothesis Set template allows for two disposition files: one containing only valid transactions, and the other containing only invalid transactions.

- Item B "Trigger Type." The three templates allow only a very limited selection of trigger types when compared with those found to be present in the case study.
- Item C "Validity Condition Type." The three templates being compared contain very little semantics in this category as can be seen by the number of zeros recorded. The Edit Transaction File PDM template does not recognize any specific condition types at all.

- Item G "Invalid Record Disposition Determination." In the two PDM templates disposition must be determined by a separate disposition test because the other two options are not included.
VALIDATE SEMANTICS APPLIED TO APPROPRIATE PDM
AND HYPOTHESIS SET TEMPLATES

<table>
<thead>
<tr>
<th>PDM Templates</th>
<th>Hypothesis Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Sort</td>
<td>Edit Trans.</td>
</tr>
<tr>
<td>File</td>
<td>File</td>
</tr>
<tr>
<td>Validate</td>
<td></td>
</tr>
</tbody>
</table>

A. Record Sources and Destinations

1. Subject Sources
   - Number: 1 1 1-2
   - Access: Seq Seq
   - Rec Types: 1 1 M
   - Group: Levels 1 0 M

2. External Object Sources
   - Number: 1 None M
   - Access: seq seq/ran
   - Rec Types: 1 M
   - Group: Levels 0 M

3. Internal Object Sources
   - Number: None None M
   - Access: None seq/ran
   - Levels: 1-2

4. Reports
   - Total Number: 1 1 1
   - Error only: 1 1 1
   - Valid only: 0 0 0
   - Combination: 0 0 0
### PDM Templates

<table>
<thead>
<tr>
<th>Edit Sort</th>
<th>Edit Trans.</th>
<th>Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>File</td>
<td>Template</td>
</tr>
</tbody>
</table>

5. Disposition Files

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temporary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Valid only</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Invalid only</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Valid-invalid</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**B. Test**

#### TEST TYPE

<table>
<thead>
<tr>
<th>TEST TYPE</th>
<th>5</th>
<th>0</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Disposition

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Group

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### TRIGGER TYPE

<table>
<thead>
<tr>
<th>TRIGGER TYPE</th>
<th>2</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Type</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Record Type

<table>
<thead>
<tr>
<th>By Field</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

2. Record Type

<table>
<thead>
<tr>
<th>By Position</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

3. Structure

| 0 | 0 | 0 |

4. Previous Test

| 0 | 0 | 1 |

5. All Recs

| 0 | 0 | 1 |

6. Combination

| 0 | 0 | 0 |

7. Field Values

| 0 | 0 | 0 |

8. Disposition By

<table>
<thead>
<tr>
<th>Group Of Tests</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>
9. Disposition By Structure 0 0 0
10. No Individual Structure 0 0 0

C. VALIDITY CONDITION TYPE

<table>
<thead>
<tr>
<th>Field Type</th>
<th>PDM Templates</th>
<th>Hypothesis Template</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Edit Sort File</td>
<td>Edit Trans. File</td>
</tr>
<tr>
<td>10 Fields of a</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Rec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 Completeness</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30 Class</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40 Range</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50 Set Membership</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>51 Compound Cond</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>52 Case</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>53 Table Linear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>54 Table Random</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>55 File Random</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>56 File Linear</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>57 88 Levels</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>60 Value Test</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>70 Duplicates</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>80 Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>90 Compound</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PDM Templates</td>
<td>Edit Sort</td>
<td>Edit Trans.</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>D. VALIDATION TEST SUBJECT ACCESS TEMPLATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Global Template</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>2. External Source Local Access Template</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>3. Within Rec Local Access Template</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>E. VALIDATION TEST OBJECT ORIGIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Fixed Value</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2. Computed</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3. In-core Structure Pattern</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4. External Structure</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### F. REPORTING

#### INVALID

1. Flagged | N | N | N
2. Set Local Msg. | N | N | Y
3. Local Print Msg | N | N | Y
4. Local Print Rec | N | N | Y
5. Delayed Msg Setting | Y | Y | Y
   Msg Print | Y | Y | Y
   Rec Print | Y | N | Y
6. Summary Msg | N | N | Y

#### VALID

1. Local Rec Print | N | N | N
2. Delayed Rec Print | N | N | N
3. No Print | Y | Y | Y

### G. INVALID RECORD DISPOSITION DETERMINATION

1. Disposition Test | 1 | 1 | 1
<table>
<thead>
<tr>
<th>PDM Templates</th>
<th>Hypothesis Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Sort File</td>
<td>Validate</td>
</tr>
<tr>
<td>Edit Trans. File</td>
<td></td>
</tr>
</tbody>
</table>

2. Validation Test  
3. Structure Validation Test

### H. DISPOSITION ACTION

1. Set Flag  
2. Write Both  
   - No Partition  
   - Partition V/IV  
   - Partition V  
   - Partition IV  
3. Write Neither  
   - Use Valid  
   - Correct  
   - Invalid  
4. Write Valid Only  
   - No Partition  
5. Write Invalid Only  
   - No Partition
Chapter 5
Organizing and Representing Template Knowledge

The semantics identified during the case study were informally clustered into individual templates. To utilize the semantic data about patterns of programming and provide an automated support environment for programming, several important issues had to be addressed. These are:

1. The organization of semantics into templates.

2. The investigation of the relationships among templates to determine how existing templates can be used to define new templates or how existing templates are related to one another.

3. The construction of a hierarchy, based on identified relationships, which allows the choice of appropriate templates for solving a problem.

4. The representation of knowledge contained in the template set in a machine processable form so that the templates can be supported in an automated environment.

5. The design of an interface that presents the appropriate template fragments to the user and interacts with the user.
5.1 Organization of the Semantics in the File/Database Processing Domain

Before one can contemplate the design of the automated support, made more intelligent by incorporating an understanding of the domain, one must first actually organize the semantic knowledge of the domain. The templates identified for the domain were described earlier in Table 9. The next step, representing the semantic knowledge is pursued later in this chapter after a discussion of relationships among templates.

5.1.1 The Relationships Among Templates

The Jackson methodology represents an early attempt to capture the essence of problems in the file/database domain. The methodology provides a notation to describe the Input/Output file structures because handling these files is the focus of problems in the domain. In the following sections semantics for the control/processing steps that might take place while solving problems will be added to this notation. The addition of problem-oriented semantics to Jackson’s notation yields many different, related templates as shown next.

At the apex of the template hierarchy (Figure 7) is a data structure based methodology such as Jackson [Jack75], or Warnier [Warn74] that contains little problem specific semantics and, consequently, has widespread applicability. Templates at the lower...
levels of the hierarchy have more semantic content, more capabilities for providing guidance and, at the same time, more specific applicability.

**Figure 7: Hierarchy of Templates**

The hierarchy of templates is not unique. There are several major dimensions along which templates can be compared and usefully organized. For example, templates can differ:
- in their use of input and output data structures,
- in the functions they perform,
- in the semantics of other templates incorporated into their definition, and
- in the degree of specificity in the semantics.

The hierarchy in Figure 7 shows the templates organized according to the input/output structures used.

The initial level in the hierarchy in Figure 7 shows a general purpose data structure oriented methodology. The second level contains the Generalized Traversal - I template which supports the traversal of any number of data structures. The second level also contains two control primitives. Level three separates table-oriented templates from file/database-oriented templates. For example, the Generalized Traversal - I template can be used to produce the Table Traversal - H template by restricting the file/database/table-oriented semantics of the Generalized Traversal - I template to only table-oriented semantics.

Given enough interaction with the user the I template can be used for the same applications as any of its subordinate templates, to produce a correct solution. However, the steps taken by the programmer would have to be much more complex and varied. On the other hand, if the user is guided to an appropriate low-level template by the environment, the user-environment interactions are simpler, standardized and more support is provided to the user.

The left subtree in the hierarchy is formed from templates that
focus on the traversal of internal data structures (e.g., Cobol tables). While the templates in the right subtree primarily manipulate external data structures. The type of data structure manipulated is not the characteristic used to differentiate among templates within either of these two subtrees. Within each subtree the nature of the processing performed or details of the implementation language used separates one template from another. A hierarchy formed using a single criteria, such as the type of processing performed, with lower-levels refining upper-levels in terms of that single criteria, has some appeal. However, the hierarchy in Figure 7 is more useful in determining, the identity of the template most appropriate for solving the problem based on some interaction with the user.

Within the H template subtree, templates that initialize/reinitialize tables are separated from those that search tables. In the initialization/reinitialization subtree the A template enters fixed values into the table, while the B and the J templates obtain values from a file to be entered into the table. The B template sequentially places data into the table while the J template places data directly.

The search branch differentiates the seldom utilized binary search template P (included for completeness) from the more common linear searches present in the F and the G templates. The difference between the F and the G templates is the use of the Cobol
verb "SEARCH" in the former and the construction of a linear search using loops in the latter.

Templates in the right and left subtrees of the I template differ in type of data structure processed and also in other ways. The templates in the left subtree must be combined with other templates from the hierarchy to be useful. They do not produce stand-alone programs. Most often their semantics are embedded in the definition of a more complex template, but they can be combined at instantiation time by the template user. Templates in the right subtree, on the other hand, can produce programs without being combined with any other template; although frequently they are used in combination with other templates. Also, the semantic content of these templates is much more extensive and complex.

The trade-off to be considered when designing a template hierarchy is between having a large number of very specific templates or fewer templates with a more general set of semantics. While, to a degree, the author opted for specific templates in the left subtree, the process was not carried to the extreme. It would have been possible to define several separate templates subordinate to the G template. The following additional semantics could have been used to generate additional subordinate templates:

1. Order of table items producing the ability to terminate a search whenever the item being searched for is greater than the current table item.

2. Identifying the number of items to be located that meet
the search criteria (i.e., only 1, or the first 2, or ...)

These variations of a linear search can be determined by simple questions and, thus, are not included as separate templates, but rather as semantic variations within a single template. Compare this with the definition of the two distinct templates F and G. While a prompt asking if the table is indexed or not decides between the use of the two templates, the code generated, and thus the prompts for information, in each case is very different. The F template uses the Cobol SEARCH verb, while the G template relies on the PERFORM-VARYING verb.

The B template also illustrates the process for forming a new template. The B template requires the nesting or combining of two templates in its definition. It contains the H template semantics for traversing a table and the I template semantics, properly restricted, for traversing a single sequential file source. Figure 8 illustrates the combining of two templates to form a third by identifying the origin of the code generation semantics present in this sample application of the B template.
Template Contributing Source Code
the Line of Code

I OPEN INPUT <sequential-file-name>
I MOVE ZERO TO <eof-flag>
I READ <sequential-file-name> AT END
I <empty file process>
I MOVE 1 TO <eof-flag>
I
H PERFORM <sequential-table-init> VARYING
H <lcv> FROM 1 BY 1 UNTIL <eof-flag> = 1
H OR <lcv> > <max-table-size>
I CLOSE <sequential-file-name>
I
H <sequential-table-init>.
H MOVE <record> TO <table-element>(lcv)
I READ <sequential-file-name> AT END
I MOVE 1 TO <eof-flag>
I

Figure 8: Generic Code Generated by the B Template
Showing the Original Source of the Semantics
1. <before_loop_process>
2. PERFORM <every_element_loop> VARYING <lcv>
   FROM 1 BY 1 UNTIL <lcv> > <maximum_value>
3. <after_loop_process>
4. ...

5. <every_element_loop>.
6. <every_element_process>
7. .

Figure 9: Generic Code Generated by the Table Traversal-H Template
1. \texttt{<before_search_process>}
2. PERFORM \texttt{<search_table_loop> VARYING <lcv>}
   FROM 1 BY 1 UNTIL <lcv> > <maximum_lcv> OR
   \texttt{<found_flag> = 1}
3.1 IF \texttt{<found_flag> = 1}
3.2 \texttt{<found_process>}
3.3 ELSE
3.4 \texttt{<not_found_process>}
4. 
5. \texttt{<search_table_loop>.}
6.1 IF \texttt{<search_criteria>}
6.2 MOVE 1 TO \texttt{<found-flag>}
7. 

\textbf{Figure 10:} Generic Code Generated by the Table Search-G Template

\textsuperscript{15}The sample generic code assumes a single-level table and a search for only the first element meeting the search criteria. These sample criteria are not restrictions of the template, but are used for presentation purposes.
Two closely related templates may differ only by a simple transformation applied at template definition time. Consider the sample generic code generated by the H template as shown in Figure 9. It contains no semantics restricting the purpose of the table traversal code contained therein. The G template is derived from the H template by a transformation that imposes a more specific purpose for the traversal. The generic code in both Figures 9 and 10 assumes a simple single-level table. Both templates are capable of dealing with much more complex tables.

The effect of the transformation used on the H template to define the G template is as follows:

- more specific information has been used in the prompts (elements contained in "< >" brackets.) For example, <every element loop> of the H template has been transformed into the more specific <search table loop> of the G template.

- additional structural "semantics" have been included. For example, the <after loop process> of H has become the IF-THEN-ELSE structure of the G template whose purpose is to deal with the results of the search.

Another example of such a mapping occurs in the Sequential Table Initialization Using a Sequential Source - B template that derives its semantics from the Table Traversal - H and the Generalized Traversal - I templates. A sample of the transformations needed is given in Figure 11.

Thus, two kinds of transformations can be used to incorporate one template into the definition of another. The first
Figure 11: Two Types of Mappings Used to Define a New Template

transformation type merely restricts the syntax and/or the semantics

16. The r mapping restricts the process action semantics of H to only the movement of record fields to table elements.

17. The s mapping both restricts the number of sources to be traversed to only 1 and restricts the access to sequential.

18. The u mapping expands the existing type_dss file-oriented semantics to include table-oriented semantics so that the relationship between table items to receive the data and file fields providing the data can be expressed.
of the contributing template. The resulting template therefore contains a subset of the original template semantics and this subset of original semantics is said to be independent of the newly defined template. The second type of transformation significantly alters the semantics of the contributing template in such a way that the resulting semantics are no longer just a special case of the original template, instead these semantics are now dependent on the newly defined template. The next section extends this classification of template semantics and describes how such a classification can be used to identify the appropriate template that is to be used.

5.2 Semantics of Individual Templates

The semantics associated with a single template are classified as essential or inessential to the template and as dependent or independent of the template.

Semantics of a template that must appear when the template is used are called essential. Both the "transaction" concept and the "master" concept are essential to the Update template. These concepts must play a role when the template is used, essential semantics are good candidates for differentiating and choosing among templates. Inessential semantics may or may not appear when the template is used and therefore are less reliable indicators of the appropriateness of applying the template to solve a problem. For
example, generation of a report in conjunction with the **Update** template is inessential. The existence of a transaction file as a source of update actions is inessential because the transactions might not require enumeration in a file.

Semantics in a template can result from redefining semantics existing in another template as described in the previous section. When the mapping does not merely restrict the semantics of another template, but actually transforms (in more sophisticated ways) the semantics into new semantics, the semantics are said to be dependent on the newly defined template. Semantics that are defined for the first time and are not the result of any transformation of existing semantics are likewise said to be dependent. Semantics inherited intact from another template and incorporated into the definition of the current template or semantics produced by a mapping that merely restricts the semantics of the original template to a special case are said to be independent because they are independent of the template being defined. They are independent of the template being defined in the sense that their origin and ownership rests with another template.

The **Sequential Table Initialization Using a Sequential Source** - B template restricts the **Generalized Traversal - I** template semantics to a single sequential file. While this restriction is essential to the B template, many other templates could process a single sequential file. The number of sources and the details of
the source access semantics are therefore independent of the B template and not reliable for predicting its use. Grouping together of templates with similar independent semantics can aid in determining the identity of the most appropriate template for a problem by narrowing the group of templates to be considered by the user.

Expansion of the generic "process action" present in the Generalized Traversal - I template into the validation test semantics present in the Validate - V template is an example of a transformation that forms dependent semantics.

This categorization of semantics into essential/inessential and dependent/independent can be further extended to include all possible combinations of the categories:

- Essential-dependent - critical and unique to this template.

- Essential-independent - critical to this template, but present in other templates also.

- Inessential-dependent - unique option or variation offered with this template.

- Inessential-independent - option or variation offered with this template, but also offered in other templates.

In attempting to identify the appropriate template to be applied to a problem, essential-dependent semantics are a prime source of criteria for differentiating among templates. Where a hierarchy of templates exist, essential-independent semantics can help narrow the search to a subtree of the hierarchy. Inessential-
dependent semantics can be used to further refine choices based on the presence of an option in the template. Inessential-independent semantics are of little use other than possibly grouping all the templates that offer a particular option.

5.3 Representing Knowledge in Templates

5.3.1 Attributed Grammar Form Model

A comprehensive formalism for representing both individual templates and the relationships among templates must allow the following:

1. representation of structural semantics,
2. representation of context-sensitive semantics,
3. flexible combination of these semantics within a template,
4. modeling of the transformations used to derive definitions of templates to show relationships among templates,
5. presentation of the template which permits the system to display prompts to elicit user-supplied application-specific data and/or source code.

The attributed grammar form model meets some of the above criteria. Thus, this model is used as a possible way to represent aspects of the templates and point to further work. For convenience, a definition borrowed from Kuo [Kuo83c] is given in Appendix F.
5.3.1.1 Traditional Use and Evaluation of Semantic Functions

Attribute grammar research has centered on the use of semantic functions for the code generation phase of the compilation process. In this role a complete parse tree is constructed where each node of the parse tree consists of the symbol from the vocabulary of the grammar, attributes of the symbol, and semantic functions for determining attribute values. Given a complete parse tree the role of identifying and scheduling semantic functions for execution is handled by the evaluator. The evaluator must determine which semantic functions are currently available to be evaluated, i.e., all needed values of parameters of the semantic function have been determined. Since any number of semantic functions spread throughout the parse tree may be ready to be executed, identifying executable semantic functions is highly complex. The set of functions which are available can change after the execution of each semantic function. This is caused by the availability of new attribute values produced by the last semantic function executed. A second problem to be dealt with by the evaluator is scheduling the order of execution among the semantic functions that are ready for execution.

The research reported here is concerned with program construction rather than program compilation. Refinement is the primary paradigm used in program construction. The grammar symbols, attributes and semantic functions are used to aid and guide the
programmer through the refinement process. Used in this generative manner the evaluator needs to evaluate semantic functions resident in only a partially completed derivation tree.

Further, the nature of the semantic functions has been expanded from their traditional role of moving data around the parse tree. In this expanded role semantic functions will not only continue to move data around the derivation tree, but will also be able to transform the attributed grammar used to represent the template. In this role semantic functions will modify the vocabulary, productions, and semantic functions of the grammar itself. Because of this expanded non-traditional role the term plan will be used to refer to what is called semantic functions in traditional uses of attributed grammars. A number of opportunities for plans local to an individual template (thus, called local-plans) have been identified and are described in Table 15.

In place of the traditional evaluator two experts might be defined. The global expert might deal with the selection of individual templates from the complete set of templates. Once a template is selected by the global expert the local expert is given control and executes the plans of a single template. Both experts are rather simple schedulers whose power and knowledge is encoded in the plans.
Table 15: Uses for Local-Plans in the Template Approach

A: - answer plan which serves as a source of answers to prompts before the user is prompted for the answer. This eliminates the need for the user to repeat application information.

C: - collection plan which traverses the derivation tree collecting attribute values to display to the user in the form of a choice list.

I: - interface plan which invokes the template-set expert at points where other templates may be embedded (with locality of semantics) within the current template. It also passes application information collected to the template-set expert.

K: - consistency plan which is responsible for enforcing consistency constraints.
Table: 15 (continued)

P: - program plan which is responsible for generating skeletal source code.

R: - refinement plan which is responsible for identifying the production or set of productions that may next be used to refine a node. It is only defined for non-terminals. It may be invoked immediately where several productions contribute to a single blank form or it may be invoked when the user returns to continue refinement of a non-terminal.

S: - semantic function which provides attribute values.

T: - tuning plan which is responsible for eliminating unnecessary productions, terminals and non-terminals and for customizing prompts.

Z: - attribute analyzer plan which is responsible for movement of attribute values around the derivation tree.
5.4 Representation of the Knowledge in the Generalized Traversal and Validate Templates

The feasibility of using an attributed grammar form to represent a simplified version of Jackson methodology has been established by Logrippo and Skuce [Logr83] and by Ramanathan and Soni in [Soni83b]. What follows is a grammar based definition of the Generalized Traversal - I template which is an enhancement and expansion of the grammar presented by Logrippo and Skuce.

5.4.1 Definition of a Preliminary Version of the I Template Grammar

The design of an attributed grammar to represent a realistic template is a complex task. The approach used herein is to first define a grammar that represents the constructs present in a Jackson program structure diagram. This grammar will be called "the I structural grammar." Next, semantic enhancements to be added will be identified. These enhancements will be included in the complete I template attributed grammar form definition.

Jackson [Jack75], motivated by Dijkstra's D-structures [Dijk68], defined a graphical notation that can be used to represent the structure of input and output files to a program. This notation allows for the following constructs:

- elementary components - not further refined.
- sequence - of two or more parts occurring once each.
A "part" is defined as any of the above constructs which, with the exception of elementary components, may themselves be composite types. Thus, nesting of types within types with the exception of elementary components is allowed.

Logrippo built his attributed grammar on the above ideas and included the following assumptions and simplifications:

1. Only the structure present in a single sequential input file would be considered.

2. The only elementary component allowed is called "simple_dss," which has to be a record type from the input file or an end of file of the input file.

3. Iteration can be accomplished on record type (i.e., a consecutive collection of records of the same type), or on groups of records with identical record types and equal key field values.

4. For key iteration groups the identity of the group (i.e., grouping value) is always supplied through the input file by the first record encountered that does not belong to the group currently being processed.

5. Any termination of an iteration group (other than the solitary end of file) leaves a record in memory ready to be processed.

6. Selection, called "choice_group", only makes selections based on record type because each "choice" begins with simple_dss which describes a record type.

Some of these restrictions are forced by the semantic functions that are used to generate code. Therefore, as a preliminary step this author defined a grammar that implemented the original Jackson
### Table 16: Productions in the I-Structural Grammar

1. `program_spec` $\rightarrow$ `type_das`

2. `type_das` $\rightarrow$ { `simple_das`
   | `choice_das`
   | `iterative_das`
   }
   { `type_das`
   | `nil`
   }

3. `simple_das` $\rightarrow$ `<label>`
   { `type_das`
   | `nil`
   }

4. `choice_das` $\rightarrow$ `choice`
   { `choice_group`
   | `nil`
   }

5. `choice` $\rightarrow$ `<choice_label>`
   { `type_das`
   | `nil`
   }

6. `choice_group` $\rightarrow$ `choice`
   { `choice_group`
   | `nil`
   }

7. `iterative_das` $\rightarrow$ `<iterative_label>`
   { `type_das`
   | `nil`
   }

---

Where:

- "<>" denote symbols to be presented to the user as prompts for application specific data or program code. Prompts require user interpretation.

- "( )" enclose a series of alternatives which are separated by ">".
constructs and ignored semantic functions (see Table 16).

Production 2 represents a choice among an elementary component, selection, or iteration. It also represents a sequence by allowing one of the three choices to be followed by type_dss. In Jackson's graphical notation simple_dss is a rectangle, choice_dss is a sequence of rectangles with circles, and iterative_dss is a rectangle with a "*".

Production 3 defines an the elementary component if "nil" is selected; otherwise, if type_dss is selected, the simple_dss is an abstraction that will be further refined. In this second case, the simple_dss acts like the "part" Jackson describes. Note productions 2, 3, 5, and 7 all provide the type_dss option to allow for nesting or refinement.

5.4.2 Enhancing the I-Structural Grammar

The following enhancements were identified as being necessary to translate the I-structural grammar into the Generalized Traversal - I template:

1. Code generation functions for simple_dss, choice_dss and iterative_dss, which are provided by the P: local-plan.

2. Multiple input and/or output file initial and final actions, which are provided by productions 2, 3, 4, 5 and 7.

3. Priming accesses for multiple input files, which are provided by production 6.

4. Multiple next accesses which may or may not be needed for
each simple_dss, which are provided by productions 9, 10 and 11.

5. A way of inheriting conditions in a nested structure, but not requiring conditions to be inherited, which is provided by the C: local-plan.

6. Several types of iterative_dss, which are provided by productions 16, 17, 18, 19 and 20.

7. Several possibilities for obtaining logical grouping criteria not just from the input file, which are provided by productions 21, 22 and 23.

8. User specification of conditions that are not limited to record type, which are provided by prompts for user code.

All of these enhancements are incorporated in the I template grammar presented next. Refer to Table 15 for definitions of the functions performed by the local-plans.

<table>
<thead>
<tr>
<th>Grammar Production</th>
<th>Local-Plans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. program --&gt;</td>
<td>S: f_cnt := 0</td>
</tr>
<tr>
<td></td>
<td>S: t_cnt := 0</td>
</tr>
<tr>
<td></td>
<td>S: db_cnt := 0</td>
</tr>
<tr>
<td></td>
<td>data_structure_environment R: #2</td>
</tr>
<tr>
<td></td>
<td>program_spec R: #4</td>
</tr>
<tr>
<td>2. data_structure_environment --&gt; Z: inherit f_cnt, t_cnt, db_cnt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>structure R: #3</td>
</tr>
<tr>
<td></td>
<td>{ data-structure_environment R: #2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3. structure --&gt;</td>
<td>S: a_t := &lt;access_type&gt;</td>
</tr>
<tr>
<td></td>
<td>S: s_id := &lt;structure_id&gt;</td>
</tr>
<tr>
<td></td>
<td>A: search data dictionary for description of s_id</td>
</tr>
<tr>
<td></td>
<td>IF found THEN</td>
</tr>
<tr>
<td></td>
<td>T: place found description in &lt;structure_defn&gt; as a choice</td>
</tr>
<tr>
<td></td>
<td>T: place found type in &lt;structure_type&gt; as a choice</td>
</tr>
</tbody>
</table>
<structure_defn>(s_defn) S: s_defn :=
<structure_defn>
<structure_type>(s_type)
S: s_type := <structure_type>
S: IF s_type = "file" THEN
  f_cnt := f_cnt + 1
ELSE IF s_type = "db" THEN
  db_cnt := db_cnt + 1
ELSE
  t_cnt := t_cnt + 1
ENDIF

4. program_spec — >
  file_init_process R: #5
  priming_access
    C: collect s_id for structure where a_t is of a type requiring priming access
    IF found THEN
      R: #6
    ELSE
      R: none
    ENDIF

type_dss
  R: #8

file_final_process
  R: #7
  S: f_f_p_cnt := 0
<cleanup_process_action>(c_p_a)
S: c_p_a := <cleanup_process_action>
P: c_p_a
P: "END"

5. file_init_process — >
  S: f_i_p_cnt := f_i_p_cnt + 1
  C: collect a_t, s_id
  T: tune the prompt using a_t, s_id
<state_id accessed as a_t initial_process_action>(f_i_p_a)
S: f_i_p_a :=
  <state_id accessed as a_t initial_process_action>
P: f_i_p_a
IF f_i_p_cnt >= f_cnt + db_cnt + t_cnt THEN
  R: none
ELSE
  R: #5
ENDIF
6. priming_access --> C: collect s_id where a_t requires
a priming access

<priming_access_for a_t>(p_a)
    T: tune the prompt with a_t
    S: p_a := <priming_access_for a_t>
    P: p_a
    C: collect s_id where a_t requires
       a priming access
       IF found THEN
          R: #6
       ELSE
          R: none
       ENDIF

7. file_final_process --> S: f_f_p_cnt := f_f_p_cnt +1
   C: collect a_t, s_id

<for s_id final_process_action>(f_f_p_a)
    T: tune the prompt with s_id
    S: f_f_p_a := <for s_id final_process_action>
    P: f_f_p_a
    IF f_f_p_cnt >= f_cnt + db_cnt
    THEN
        R: none
    ELSE
        R: #7
    ENDIF

8. type_dss -->
   { simple_dss R: #9
     | choice_dss R: #12
     | iterative_dss R: #16
   }
   { type_dss R: #8
     | nil
   }

9. simple_dss -->
   { access_simple_dss R: #10
     | no_access_simple_dss R: #11
   }
   { type_dss R: #8
     | nil
   }

10. access_simple_dss -->
<label> (L) S: L := <label>
<process L>(p_l) T: insert L into the prompt
S: p_l := <process L>
P: p_l

/* For process paragraphs with remote bodies a
stacking plan could be implemented at this
point and similar points */

<navigation> (n_a) C: collect n_a
T: display n_a
S: n_a := <navigation>
P: n_a

11. no_access_simple_dss -->

12. choice_dss -->
choice
{ choice_group
| default_choice
| nil

13. choice -->

{( <choice_label>(c_l) S: c_l := <choice_label>
<choice_condition>(c_cond)
C: collect all "cond" type attributes
in the hierarchical path of this
node and display them as choices.
S: c_cond := <choice_condition>
P: c_cond

{( <process c_l>(p_c_l) T: insert c_l into prompt
S: p_c_l := <process c_l>
P: p_c_l

{ type_dss
| nil
}
)
| nil

P: "NEXT SENTENCE"
14. choice_group --> P: "ELSE IF"
   choice R: #13
   { choice_group R: #14
       | default_choice R: #15
   }

15. default_choice -->
   <default_choice_label>(d_c_l)
   S: d_c_l := <default_choice_label>
   P: "ELSE"

   {{ <process d_c_l>(p_d_c_l)
     T: insert d_c_l into prompt
     S: p_d_c_l := <process d_c_l>
     P: p_d_c_l
     {
       type_dss R: #8
       | nil
     }
   }
   | nil
   P: "NEXT SENTENCE"
}

16. iterative_dss -->
   <iterative_label>(i_l)
   S: i_l := <iterative_label>
   { conditional_iterative_dss R: #17
     | count_iterative_dss R: #19
     | key_iterative_dss R: #20
   }

17. conditional_iterative_dss -->
   Z: inherit i_l from parent
   T: insert i_l into prompt
   P: "DO WHILE"

   <conditional_iterative_condition_for i_l>(c_i_cond)
   C: collect all cond attributes in the
   hierarchical path and display as
   choices.
   S: c_i_cond :=
   <conditional_iterative_condition_for i_l>
   P: c_i_cond
   {
     type_dss R: #8
     | nil
   }
   end_iterate R: #15
18. end_iterate —> Z: inherit i_l from parent
T: insert i_l into prompt

{ <next_access_for i_l>(n_a)
  C: collect and display all n_a
  S: n_a := <next_access_for i_l>
  P: n_a
| nil
}

P: "END DO"
Z: inherit i_l from parent
T: insert i_l into prompt

<process_completed i_l>(p_c_i_l)
  S: p_c_i_l := <process_completed i_l>
  P: p_c_i_l

19. count_iterative_dss —> Z: inherit i_l

<loop_count_variable>(1cv)
  S: 1-c-v := <loop_count_variable>
  <count_initial>(ci_init)
    S: ci_init := <count_initial>
    <count_increment>(ci_inc)
      S: ci_inc := <count_increment>
      T: insert i_l, l_c_v, ci_init, ci_inc into code.
      P: "DO FOR l_c_v = ci_init BY ci_inc WHILE"
  <count_iterative_condition>(cnt_i_cond)
    S: cnt_i_cond := <count_iterative_condition>
    P: cnt_i_cond

{ type_dss R: #8
  | nil
} end_iterative R: #15

20. key_dss —> S: e_k_g_cond := null
Z: inherit i_l

establish_key_grouping_criteria R: #21
  Z: obtain synthesized e_k_g_cond
  P: "DO WHILE"
<key_iterative_condition_for i_l>(k_i_cond)
  T: insert i_l into prompt
  C: collect all "cond" type attributes
    in the hierarchical path of this
node and display as choices.

S: $k_i$ _cond :=
    <key_iterative_condition_for i_1>

P: $k_i$ _cond
P: $e_kg$ _cond

{ type_dss
 | nil
}
end_iterative

21. establish_key_grouping_criteria -->

{ field_from_current_record
    R: #22
| field_from_another_structure
    R: #23
| <other_process_action>(o_p_a)
    S: o_p_a := <other_process_action>
}

{ establish_key_grouping_criteria
    R: #21
    S: $e_kg$ _cond := append($e_kg$ _cond, "AND")
    | nil
}

22. field_from_current_record -->

    Z: inherit $e_kg$ _cond
    <field_name>(f_n)
    S: f_n := <field_name>
    T: insert f_n into code
    P: SAVE-f_n := f_n

    <relational_operator>(r_o)
    S: r_o := <relational_operator>
    S: $e_kg$ _cond :=
        append($e_kg$ _cond,SAVE-f_n r_o f_n)

{ field_from_current_record
    R: #22
    S: $e_kg$ _cond := append($e_kg$ _cond,"AND")
    | nil
}

23. field_from_another_structure -->

    <process_to_produce_group_identity_code>
        (p_t_p_g_i_c)
    Z: inherit $e_kg$ _cond
    S: p_t_p_g_i_c :=
        <process_to_produce_identity_code>

    <field_from_another_structure_condition>
        (f_f_a_s_cond)
5.4.3 The Process of Creating a Template Representation

Representing the semantics of a problem class as a template definition is a complex task as can be seen by the details contained in the I template definition. Since the I template semantics dealing with the traversal of arbitrarily complex input and output structures is the foundation of other templates, it is natural to look for a systematic way of transforming the I template definition to produce the definition of other templates. This process must incorporate the additional semantics defined for the new template within the traversal semantics captured in the I template representation. This approach was used to construct the representation of the Validate template. The author found the I template definition to be very helpful in constructing the syntactic aspects of the Validate template definition, but less helpful in constructing the necessary local-plans for Validate.

A representative example of the relationships among the I and Validate template productions is given in Table 17. The laborious manual creation of both the I and Validate template definitions
highlighted the need for automated support of this task. As a first step in the design of such support, this author defined the nature of transformations encountered in the construction of the Validate template definition. Further research remains to be done to established that the attributed grammar form model is adequate for describing such a set of transformations.

The definition of a new template can be obtained by transforming existing template(s) in one or several of the following ways.

Restriction.

1. Restricting the semantics of an existing template so that the new template only applies to a subset of the problems covered by the original template. For example, V restricts I semantics to structures serving five specific functions.

String Substitution.

2. Adding meaning to existing prompts, which is possible because more is known about the problems to be addressed. For example, the "<s_id accessed as a_t initial_process_action>" prompt of the I template becomes the "<subject_record_source_init_process_action>" of the V template.

3. Expanding a single general prompt into several prompts containing more semantic information specific to the new subset of problems. For example, the prompt <s_id accessed as a_t initial_process_action> of the I template becomes "<subject_record_source_init_process_action>, "<external_object_source_init_process_action>, "<internal_object_init_process_action>, "<report_init_process_action>," and "<disposition_init_process_action>" of the V template.
Table 17: Relationships Among I Template and Validate Template Productions

<table>
<thead>
<tr>
<th>I Template Production</th>
<th>Related V Productions</th>
<th>Nature of Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Program specified as validate.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Five specific structures are identified for Validate.</td>
</tr>
<tr>
<td>3</td>
<td>3,4,5,6,7</td>
<td>Structure information is now identified by the purpose the structure serves.</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>Initial and final process actions are identified by structure purpose. The test_entry nonterminal is introduced to permit tests to be executed after inputs are processed.</td>
</tr>
<tr>
<td>5</td>
<td>9,10,11,12</td>
<td>Expand to allow for each of the possible Validate structure types.</td>
</tr>
<tr>
<td>7</td>
<td>17,18,19,20</td>
<td>Expand final action to prompt for final action by structure type.</td>
</tr>
<tr>
<td>10,11</td>
<td>15,16</td>
<td>Include the new nonterminal test_entry which when refined incorporates the testing semantics of Validate.</td>
</tr>
<tr>
<td>33 through 56</td>
<td></td>
<td>New productions not present in any form in the I template.</td>
</tr>
</tbody>
</table>
Embedding One Template in Another.

4. Identifying opportunities for combining templates. For example, "<process L>" of the I template expands to the nonterminal "test_entry" of the V template allowing for local access to tables or files to obtain test objects.

Transformation of Semantics.

5. Changing attributes and corresponding semantic functions. For example, the "f_cnt" attribute that counts the number of files in the I grammar by semantic function "S: f_cnt := f_cnt + 1" needs to be expanded to keep count of files by purpose served in the V template.

Optimization.

6. Overcoming the problem of code migration that can occur when templates are nested by having the human expert define the template construct in the appropriate productions.

5.5 Attribute Grammar for the Validate - V Template

Chapter 7 contains a scenario of the system/user interaction as well as attributes and plans for the Validate template productions as they apply to the scenario. The grammar given here for the Validate template has limited the local-plans to code generation (expressed as "-[.....]").
{ -[<internal_object_record_source_init_process_action>]
  | choose_A,B,J_or_R_templates
}  
{ internal_object_record_source_init
  | nil
}

11. report_init —> 
{ -[<report_init_process_action>]
  | nil
}  
{ report_init /* Semantics from the X template can be
  incorporated into the definition at
  this point by the human designer*/
  | nil
}

12. disposition_file_init —> 
{ -[<disposition_file_init_process_action>]
  | nil
}  
{ disposition_file_init
  | nil
}

13. type_das —> 
{ simple_das
  | choice_das
  | iterative_das
}  
{ type_das
  | nil
}

14. simple_das —> 
{ access_simple_das
  | no_access_simple_das
}  
{ type_das
  | nil
}

15. access_simple_das —> 
<label>
{ test_entry
  | nil
}  
-[<process L>]
-[<next_access>]

16. no_access_simple_das —> 
<label>
{ test_entry
  | nil
17. external_object_record_source_final -->
   -[<external_object_record_source_final_process_action>]
   { external_object_record_source_final
    | nil }

18. internal_object_record_source_final -->
   -[<internal_object_record_source_final_process_action>]
   { internal_object_record_source_final
    | nil }

19. report_final -->
   -[<report_final_process_action>] /* The designer could
   insert X template
   semantics here */

   { report_final
    | nil }

20. disposition_file_final -->
   -[<disposition_file_final_process_action>]
   { disposition_file_final
    | nil }

21. choice_dss -->
   -[IF]
   choice
   { choice_group
    | default_choice
    | nil }
   -[ENDIF]

22. choice -->
   <choice_label>
   -[<choice_condition>]
   { test_entry
    | nil }
   {{ -[<process L>]
      { type_dss
       | nil }
     }
   | -[NEXT SENTENCE]}

23. choice_group -->
-ELSE IF
choice
{ choice_group
  | default_choice
}

24. default_choice —>
<default_choice_label>
-ELSE
{ test_entry
  | nil
}
{ ( -[process 1>
  { type_dss
    | nil
  }
  | -[NEXT SENTENCE]
}

25. iterative_dss —>
<iterative_label>
{ conditional_iterative_dss
  | count_iterative_dss
  | key_iterative_dss
}

26. conditional_iterative_dss —>
-[DO WHILE]
-[:conditional_iterative_condition>]
{ test_entry
  | nil
} { type_dss
  | nil
} end_iterative

27. end_iterative —>
{ -[<next_access>]
  | nil
} -[ENDDO]
{ test_entry
  | nil
} -[<process_completed l_1 >]

28. count_iterative_dss —>
<loop_count_variable>
<count_initial>
<count_increment>
-[DO i_1 FOR lc_v = ciint BY ciinc WHILE]
29. key_dss —> establish_key_group_identity
   [DO WHILE]
   [<key_iterative_condition>]
   [<e_k_g_cond>]
   { test_entry
    | nil
   }
   { type_dss
    | nil
   }
end_iterative (L)

30. establish_key_group_identity —> 
    { field_from_current_record
    | field_from_another_structure
    | [<other_process_action>]
    } 
    { establish_key_grouping_criteria
    | nil
   }

31. field_from_current_record —> 
    <field_name>
    <relational_operator>
    [SAVE_fn := fn]
    { field_from_current_record
    | nil
   }

32. field_from_another_structure —> 
    [<process_to Produce_identity_code>]
    { field_from_another_structure_cond
    | nil
   }
    { field_from_another_structure
    | nil
   }

The following productions offer refinements to non-terminal symbols introduced by translating the I grammar into the V grammar.

33. test_entry —> 
    { validation_test
    | process_test
}
34. validation_test -->
    test_trigger
    { local_subj ect_traversal
    | nil
    } local_object_traversal
    { -{<local_test_preparation_process_action>]
    | nil
    } validity_test
    -[ENDIF]
35. test_trigger -->
    -[IF]
    trigger_type
    -[tt trigger_condition>]
36. validity_test -->
    -[IF]
    validity_test_type
    -[<validity_condition>]
    { valid_record_process_action
    | -[NEXT SENTENCE]
    }
    -[ELSE]
    { invalid_record_process_action
    | -[NEXT SENTENCE]
    }
37. process_test -->
    -[IF]
    -[<process_test_condition>]
    --[<process_test_process_action>]
    -[ENDIF]
38. disposition_test -->
    -[IF]
    disposition_test_type
    -[<disposition_condition>]
    true_disposition_process
    -[ELSE]
    false_disposition_process
    -[ENDIF]
39. true_disposition_process -->
disposition_process

40. false_disposition_process —> disposition_process

41. disposition_process —> { -<write_valid_and_invalid_process_action> ]
| -<write_valid_only_process_action> ]
| -<write_invalid_only_process_action> ]
| nil /* i.e. write neither*/
} 
| -<reset rvtrf flag_action> ]
| nil /* the flag may be needed in another test*/

42. disposition_test_type —> 
| by_group_of_validation_test_results
| by_structure_validity
| other

43. report_test —> 
| [IF]
| [-<report_condition>] true_report_test_action
| [ELSE]
| false_report_test_action
| [ENDIF]

44. true_report_test_action —> report_test_action

45. false_report_test_action —> report_test_action

46. report_test_action —> 
| ([ { -<print_valid_record>
| nil
} ]
| ([ { -<print_error_message_process_action>
| nil
} ]
| [-<print_invalid_record_fields_process_action>
| nil
} ]
| [-<reset rvtrf flag_action>]
| nil /* the flags may still be needed*/

47. local_subject_traversal —>
/* include the I template grammar appropriately mapped for
nesting (for example, eliminate initial process actions, execution ending statements, etc) */

48. local_object_traversal — >
/*include the F, G, P, H, and I templates appropriately mapped*/

49. trigger_type — >
{ record_type_determined_by_record_fields
  | record_type_determined_by_record_position
  | type_of_data_structure
  | previous_validation_test_results
  | execute_on_all_records
  | record_field_values
  | validity_of_structure
  | other
}
{ trigger_type /* allow for combinations */
  | nil
}

50. validity_test_type — >
{ between_fields_of_the_current_record
  | field_completeness
  | class_of_data
  | range_of_data
  | set_membership
  | fixed_value_test
  | duplication_test
  | group_structure_test
}
{ validity_test_type
  | nil
}

51. valid_record_process_action — >
{ /*[NEXT SENTENCE]
  |{ valid_record_report_action
    | nil
  }
  { disposition_action
    | nil
  }
  { /*<other_valid_record_process_action>*/
    | nil
  }
}

52. disposition_action — >
{ /*<set_record_error_flag>*/
  | nil
}
{ /*<write_valid_and_invalid_process_action>*/
| ![write_valid_only_process_action] |
| ![write_invalid_only_process_action] |
| delay_record_disposition_action |
| nil /* i.e. write neither */ |

53. valid_record_report_action — >
{ -[<print_valid_record>]
| delay_reporting_valid |
| nil |
} /* additional report template semantics can be included at this point by introducing nonterminals */

54. invalid_record_report_action — >
{ -[<set_error_message>]
| ![print_error_message_process_action]
| ![print_invalid_record_fields_process_action]
| delay_reporting_error_message |
| delay_reporting_invalid_record_fields_process_action |
| invalid_record_report_action |
| nil |
}

55. invalid_record_process_action — >
{ remember_validity_test_results |
| nil |
} ![NEXT SENTENCE]
{ invalid_record_report_action |
| nil |
} disposition_process_action |
| delay_record_disposition_action |
| ![other_invalid_process_action] |
| nil |
}

56. remember_validity_test_results — >
<remember_validity_test_results_flag_id> — [<set_rvtrf_action>]
5.6 The Use of Artificial Intelligence Techniques

This chapter has suggested, in very specific ways, how to combine the approaches of software engineering and artificial intelligence. These are:

- Immersion in a domain: the author has attempted to categorize the tasks that a programmer goes through in order to develop programs in the domain.

- Classification of knowledge: the author has identified a conceptual model which was used to classify the tasks into distinct templates. Further, semantics are classified as essential/inessential and dependent/independent depending on their function in the template.

- Representation of knowledge: the knowledge gleaned through the case study has been partially represented using a grammar-based model. Limitations of grammar models have been identified.

- Presentation of knowledge: knowledge represented in templates is presented to the user by the template-set expert and the template expert.

The last two items are further discussed below.

The attributed grammar form model that was used here in an attempt to represent templates has many conceptual similarities with the frames approach to knowledge representation. The symbols of a production can be viewed as frame types; the attributes of the symbol, as the slots; and the plans, as procedural components associated with frames. In addition to representing the template knowledge, presentation experts have to be designed to present only the requisite amount of knowledge to the user - no more, no less.

The volume of information in the knowledge base can amount to a few
thousand productions. Due to the limited capabilities of humans, the presentation expert must reduce the number of productions visible to the user. The presentation expert can be designed, in future work, along the lines suggested below:

- The experts can be hierarchically organized depending on the scope of templates to be manipulated, i.e., the entire template set or only a single template.

- A function of the template-set expert must be to select the most appropriate template to be used.

- The template expert must interface with the template-set expert in order to bring into play any additional templates.

- The template expert must also be able to pass any application information collected, to the template set expert to aid in making template selection.

- The template expert must be able to tune the copy of the template it is currently working with to reflect current application knowledge.
Chapter 6

Application of Several Techniques to a Problem for the Purpose of Comparison

Having defined two templates (Generalized Traversal and Validate) in Chapter 5, this chapter will apply each template, the Logrippo Grammar, and the Jackson methodology to a single set of problem specifications. In particular, the ability of each technique to guide the user to a solution, the ability to generate (or suggest in the case of Jackson) skeletal code for the problem will be compared. Section 6.1 contains problem specifications for a validate application. Section 6.2 shows the use of the Logrippo grammar in solving the problem. Section 6.3 shows the use of the Jackson methodology in solving the problem. Section 6.4 shows the effects of using the Generalized Traversal template in solving the problem. Section 6.5 similarly shows the use of the Validate template.
6.1 Specifications for the Problem to be Solved Using Each Technique

OVERVIEW OF THE PROBLEM

The problem is to validate records from an input file (SUBJECT-RECORD-SOURCE) by applying several tests. Valid records (those passing all of the tests) are to be separated from invalid records (those failing one or more tests). Two output files are used to accomplish this separation, with valid records being written to the VALID-RECORD-DISPOSITION-FILE and invalid records written to the INVALID-RECORD-DISPOSITION-FILE. An error report that lists the error messages for an individual invalid record followed by a printout of the invalid record is also created. Figure 12 gives the file environment of the problem which is followed by a listing of the problem requirements using the validate problem class semantics.

6.1.1 Problem Requirements in Terms of Validate Semantics

- **Source of Record Subjects** - Two record types exist. The first record in the file is called "HEADER-TYPE". The remainder of the records are called "DETAIL-TYPE".

- **Destination of Tested Records** - Invalid records, those that fail one or more validation tests are to be written to the "INVALID-RECORD-DISPOSITION-FILE". Valid records are to be written to the "VALID-RECORD-DISPOSITION-FILE".

- **Test Types** - Three validity conditions are used: field completeness, class test, and set membership using compound condition. One disposition test that uses a group of test results condition is needed. A report test will control printing of the invalid record contents.

- **Test Triggers** for the 3 validation tests are, in order:
determined by record type (DETAIL-TYPE only), determined by record field values, and determined by previous validation test results. The disposition test and the report test are executed on all DETAIL-TYPE records.

- **External source of test objects** - none.

- **Internal source of test objects** - none.

- **Report generation** - Error message printing is controlled by individual validation tests. The printing of invalid record contents is controlled by the report test.

- **Disposition of tested records** - Both valid and invalid records are written with the valid partitioned in the VALID-RECORD-DISPOSITION-FILE and the invalid records partitioned in the INVALID-RECORD-DISPOSITION-FILE. A disposition test using a group of validation test results
condition will determine disposition of the tested records.

6.2 Logrippo Technique Applied to the Problem

Logrippo bases his technique on the structure of a single input file. This means that program statements (control and process code) dealing with the output file will have to be added by the programmer after automatic code generation is completed. Logrippo's system was not intended for production use but its application to this problem is a good benchmark of what his system could accomplish.

6.2.1 Skeletal Code Generated by the Logrippo Technique

In this and other sections showing skeletal code, comments enclosed in "/* ... */" have been added by this author and would not be produced by the technique. Prompts generated or suggested by the technique for user-supplied problem-specific information will be shown enclosed in "< >." Where the user responds to a prompt, the prompt will not appear but the users response in quotes will appear. The skeletal code generated using the Logrippo technique is in the same language used in his paper [Logr83].
BEGIN
READ;
/*SUBJECT-RECORD-SOURCE-FILE*/
DO "NOT-EOF-TYPE" ->
/*Logrippo's syntax for a DO WHILE
loop with condition NOT-EOF-TYPE*/
PROCESS "HEADER-TYPE";
READ
DO "DETAIL-TYPE" ->
PROCESS "DETAIL-TYPE";
READ
OD
/*used to delimit DO loop*/
OD;
PROCESS "EOF"
END

6.2.2 Discussion of Logrippo Results

The code generated, though not extensive, can be expanded to produce a solution. It allows for loops, controlled by record type, of which three types are present: "NOT-EOF-TYPE, HEADER-TYPE and DETAIL-TYPE." Only the first and last record types control loop execution. Accesses to the input file are correctly placed within
the skeletal control code. What is missing and must be supplied manually by the programmer is the testing of records, generation of error messages, and partitioning of records by validity. This is caused by the lack of knowledge of validate semantics in his attribute grammar.

In terms of directing the user, Logrippo's system does ask for type of structure wanted — (simple_dss, choice_dss, or iterative_dss). It also prompts for record type identity that is used in loop and case statements as conditions. Because of a lack of knowledge of validate semantics the system is unable to provide the user, help or direction in terms of the validate problem beyond the structural details related to the single input file given above.

6.3 Jackson Methodology Applied to the Problem

The Jackson methodology first requires the structure present in each input and output to be represented in individual data structure diagrams. Figures 13, 14, 15, and 16 are the data structure diagrams for the sample problem being discussed.

The nodes of the data structure diagrams are then combined into a single program structure diagram. The program structure diagram consists of named nodes arranged in a hierarchy that records sequence, multiple occurrences, and choices. The program structure diagram generated using the methodology is not unique; although, hopefully, there is less variation present when the methodology is
Figure 13: SUBJECT-RECORD-SOURCE Data Structure Diagram
Figure 14: ERROR-REPORT Data Structure Diagram

correctly applied than without the methodology. Figure 17 is one of
the possible program structure diagrams that could result from the
combination of the data structure diagrams. The programmer uses the
program structure diagram as a design for the skeletal code and
manually generates the skeletal code. Such skeletal code manually
produced for the problem specifications is shown in the next
section.
VALID-RECORD-DISPOSITION-FILE

| valid-detail-record |

Figure 15: VALID-RECORD-DISPOSITION-FILE Data Structure Diagram

INVALID-RECORD-DISPOSITION-FILE

| invalid-detail-record |

Figure 16: INVALID-RECORD-DISPOSITION-FILE Data Structure Diagram
Figure 17: Program Structure Diagram for the Sample Problem
6.3.1 Skeletal Code Suggested by the Jackson Methodology

DO WHILE <condition for "NOT-EOF-LOOP">
    PROCESS "HEADER TYPE"

    DO WHILE <condition for "DETAIL-VALIDATE TYPE">
        /*Posit 1: the detail records are all valid*/
        PROCESS "TEST1"
        PROCESS "TEST2"
        PROCESS "TEST3"
        IF <check posit 1 condition>
            PROCESS "VALID-RECORD"
        ELSE
            /*Admit: the record is invalid*/
            <set invalid record flag action>
        ENDIF
        IF <condition for "invalid record">
            PROCESS "REPORT-INVALID-RECORD"
            PROCESS "DISPOSE-OF-INVALID-RECORD"
            <reset invalid record flag action>
        ENDIF
    ENDDO
ENDDO
TEST1.
/*Posit 2: the detail record is valid for test 1*/
IF <condition for "TEST 1">
   NEXT SENTENCE
ELSE
   /*Quit action for Posit 2*/
   PROCESS "ERROR-1-FOR-REPORT"
ENDIF

TEST2.
/*Posit 3: the detail record is valid for test 2*/
IF <condition for "TEST 2">
   NEXT SENTENCE
ELSE
   /*Quit action for Posit 3*/
   PROCESS ERROR-2-FOR-REPORT
ENDIF

TEST3.
/*Posit 4: the detail record is valid for test 3*/
IF <condition for "TEST 3">
   NEXT SENTENCE
ELSE
   /*Quit action for Posit 4*/
   PROCESS ERROR-3-FOR-REPORT
ENDIF
6.3.2 Discussion of Jackson Results

Jackson requires, other than a description of the data structures, the labeling of nodes of the program structure diagram. These labels are shown enclosed in quotes in the skeletal code. The structure of both the input file and all three output files is represented in the skeletal code. While conditions are not specifically requested by Jackson, it is assumed that each choice denoted by a "o" in the program structure diagram and each iteration denoted by a "x" require conditions. Unlike Logrippo's methodology which required the inheritance of key conditions when iterations were nested, Jackson's methodology requires no such information. Access statements for the input file are not included and must be supplied by the programmer.

This example, in fact all problems dealing with validation of records, will require the use of the "backtracking" technique - a fact that the programmer is responsible for knowing. In applying backtracking, the programmer has ample opportunity to follow a unique solution and, thus, the scope for variety in the design process is greatly increased. This is caused by the need to decide which assumptions (called posits by Jackson) to make, the order of posits, the testing of posits and the nature of the process needed to back out of an incorrect assumption, i.e., quit process. Again, as with Logrippo, Jackson contains no specific validate semantics; therefore, the programmer (as far as the methodology is concerned)
approaches each validate problem using the input/output structure only. The need for backtracking, the nature and order of posits and quit processes must be rediscovered.

6.4 Generalized Traversal - I Template Applied to the Problem

The Generalized Traversal - I template, like the Jackson methodology, is based on the structure present in both the input and output files. It would be accurate to call the I template a version of the Jackson methodology that makes use of problem information collected from the user to generate code, supplies meaningful prompts for the information, presents choices to the user, and modifies the template to remain current with information collected. Both the I template and Jackson methodology rely on the programmer for forming or having knowledge of a program structure diagram. Neither contain semantics of a specific problem class as they both apply to the entire problem domain.
6.4.1 Skeletal Code Generated by the I Template

The skeletal code is preceded by the I template production number that produced the line of code. To view the production refer to the definition of the I template in the previous chapter.

<table>
<thead>
<tr>
<th>Production Number</th>
<th>Skeletal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>&lt;&quot;SUBJECT-RECORD-SOURCE-FILE&quot; accessed as &quot;INPUT&quot; initial_process_action&gt;</td>
</tr>
<tr>
<td>6</td>
<td>&lt;priming_access_for &quot;SUBJECT-RECORD-SOURCE-FILE&quot;&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;&quot;ERROR-REPORT&quot; accessed as &quot;OUTPUT&quot; initial_process_action&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;&quot;VALID-RECORD-DISPOSITION-FILE&quot; accessed as &quot;OUTPUT&quot; initial_process_action&gt;</td>
</tr>
<tr>
<td>5</td>
<td>&lt;&quot;INVALID-RECORD-DISPOSITION-FILE&quot; accessed as &quot;OUTPUT&quot; initial_process_action&gt;</td>
</tr>
<tr>
<td>17</td>
<td>DO WHILE &lt;conditional_iterative_condition_for &quot;NOT-EOF-TYPE&quot;&gt;</td>
</tr>
<tr>
<td>10</td>
<td>PROCESS &quot;HEADER-TYPE&quot;</td>
</tr>
<tr>
<td>10</td>
<td>&lt;next_access&gt;</td>
</tr>
<tr>
<td>17</td>
<td>DO WHILE &lt;conditional_iterative_condition_for &quot;DETAIL-VALIDATE-TYPE&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>/<em>Posit 1: the detail records are all valid</em>/</td>
</tr>
<tr>
<td>11</td>
<td>PROCESS &quot;TEST1&quot;</td>
</tr>
<tr>
<td>11</td>
<td>PROCESS &quot;TEST2&quot;</td>
</tr>
<tr>
<td>11</td>
<td>PROCESS &quot;TEST3&quot;</td>
</tr>
<tr>
<td>12,13</td>
<td>IF &lt;check valid-record-condition&gt;</td>
</tr>
<tr>
<td>13</td>
<td>PROCESS &quot;VALID-RECORD&quot;</td>
</tr>
<tr>
<td>15</td>
<td>ELSE</td>
</tr>
<tr>
<td>15</td>
<td>/<em>Admit: the record is invalid</em>/</td>
</tr>
<tr>
<td>15</td>
<td>&lt;set invalid record flag action&gt;</td>
</tr>
<tr>
<td>12</td>
<td>ENDIF</td>
</tr>
<tr>
<td>12,13</td>
<td>IF &lt;condition for &quot;invalid record&quot;&gt;</td>
</tr>
<tr>
<td>15</td>
<td>PROCESS &quot;REPORT-INVALID-RECORD&quot;</td>
</tr>
<tr>
<td>11</td>
<td>PROCESS &quot;DISPOSE-OF-INVALID-RECORD&quot;</td>
</tr>
<tr>
<td>11</td>
<td>&lt;reset invalid record flag action&gt;</td>
</tr>
<tr>
<td>12</td>
<td>ENDIF</td>
</tr>
</tbody>
</table>
<next_access_for "DETAIL-VALIDATE-TYPE">
ENDO
ENDO

"SUBJECT-RECORD-SOURCE-FILE" final_process_action
"ERROR-REPORT" final_process_action
"VALID-RECORD-DISPOSITION-FILE" final_process_action
"INVALID-RECORD-DISPOSITION-FILE" final_process_action

cleanup_process_action
END

TEST1.

/* Posit 2: the detail record is valid for test 1*/

12,13 IF <condition for "TEST 1">
13 NEXT SENTENCE
15 ELSE
/* Quit action for Posit 2*/
15 PROCESS "ERROR-1-FOR-REPORT"
12 ENDIF

TEST2.

/* Posit 3: the detail record is valid for test 2*/

12,13 IF <condition for "TEST 2">
13 NEXT SENTENCE
15 ELSE
/* Quit action for Posit 3*/
15 PROCESS ERROR-2-FOR-REPORT
12 ENDIF
10 TEST3.
    /*Posit 4: the detail record is valid for test 3*/
12,13 IF <condition for "TEST 3">
13    NEXT SENTENCE
15    ELSE
       /*Quit action for Posit 4*/
15      PROCESS ERROR-3-FOR-REPORT
12    ENDIF
6.4.2 Discussion of I Template Results

The I template results are very similar to those for Jackson. Improvements exist in the area of control code generation. Initial and final file actions are asked for by the system thus providing code location help. Input file "priming access" and "next access" locations are displayed for the user. What is not apparent is the helpful interaction given by the system, such as displaying all attributes from the current derivation tree that are of type "n_a" (next access). These will contain next access statements already resident in the derivation tree. This will permit the user to choose a previously written "next access" code sequence or, if none suit, to construct his own. Conditions are handled in a similar manner so that the programmer can decide if a condition should be inherited in a nested structure or not.

In the area of validate semantics the I template must rely on "backtracking" as did Jackson. This is because the I template does not know about validate. The solution is therefore based on file structure without the benefit of a validate pattern.

6.5 Validate - V Template Applied to the Problem

The Validate template, when compared with the other methods, knows the most about validate problems. It should, therefore, offer the most guidance to the programmer.
6.5.1 Skeletal Code Generated by the Validate Template

The skeletal code is preceded by the \textbf{V} template production number that produced the line of code. To view the production refer to the definition of the \textbf{V} template in the previous chapter.

<table>
<thead>
<tr>
<th>Production Number</th>
<th>Skeletal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 &lt;subject_record_source_init_process_action&gt;</td>
<td>/* For invalid records*/</td>
</tr>
<tr>
<td>8 &lt;subject_record_source_priming_access_code&gt;</td>
<td></td>
</tr>
<tr>
<td>11 &lt;report_init_process_action&gt;</td>
<td></td>
</tr>
<tr>
<td>12 &lt;disposition_file_init_process_action&gt;</td>
<td></td>
</tr>
<tr>
<td>12 &lt;disposition_file_init_process_action&gt; /* For valid records*/</td>
<td></td>
</tr>
<tr>
<td>26 DO WHILE &lt;conditional_iterative_condition_for &quot;NOT-EOF-TYPE&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>15 PROCESS &quot;HEADER-TYPE&quot;</td>
<td></td>
</tr>
<tr>
<td>15 &lt;next-access&gt;</td>
<td></td>
</tr>
<tr>
<td>26 DO WHILE &lt;conditional_iterative_condition_for &quot;DETAIL-VALIDATE-TYPE&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>35 IF &lt;record_type_determined_by_position validation_test_trigger_condition&gt;</td>
<td></td>
</tr>
<tr>
<td>36 IF &lt;field_completeness validity_condition&gt;</td>
<td></td>
</tr>
<tr>
<td>36 NEXT SENTENCE</td>
<td></td>
</tr>
<tr>
<td>36 ELSE</td>
<td></td>
</tr>
<tr>
<td>56 &lt;set remember_validity_test_results_flag action&gt;</td>
<td></td>
</tr>
<tr>
<td>54 &lt;set_error_message&gt;</td>
<td></td>
</tr>
<tr>
<td>54 &lt;print_error_message_process_action&gt;</td>
<td></td>
</tr>
<tr>
<td>34 ENDIF</td>
<td></td>
</tr>
</tbody>
</table>
35 IF <record_field_value validation_test_trigger_condition>
    36 IF <class_test validation_condition>
        NEXT SENTENCE
    36 ELSE
        56 <set remember_validity_test_results_flag action>
        54 <set_error_message>
        54 <print_error_message_process_action>
    34 ENDIF
35 IF <previous_validation_test_results validation_test_trigger_condition>
    36 IF <set_membership_using_compound_condition validity_condition>
        NEXT SENTENCE
    36 ELSE
        56 <set remember_validity_test_results_flag action>
        54 <set_error_message>
        54 <print_error_message_process_action>
    34 ENDIF
38 IF <group_of_validation_test_results disposition_condition>
    41 <write_invalid_only process_action>
38 ELSE
    41 <write_valid_only process_action>
38 ENDIF
IF <report_test condition>

<print_invalid_record_fields process_action>

<reset remember_validity_test_results_flag action>

ELSE

NEXT SENTENCE

ENDIF

<process subject_source_record>

<next_access_for "DETAIL-VALIDATE-TYPE">

ENDDO

<process_completed "DETAIL-VALIDATE-TYPE">

ENDDO

<process_completed "NOT-EOF-TYPE">

<subject_record_source_final_process_action>

<report_final_process_action>

<disposition_file_final_process_action>

/* For invalid records*/

<disposition_file_final_process_action>

/* For valid records*/

<cleanup_process_action>

END
6.5.2 Discussion of Validate Template Results

The Validate template shares all of the advantages that the I template did over the other methods along with some major additions. The Validate template has incorporated into its definition, semantics of the problem class. It therefore knows about the subject record source file, external and internal object sources, report and disposition files. Validate uses this knowledge to direct the user to name the "subject-record-source-file" since that input is essential to the problem class. It also knows to ask if the other files are present or not, since they are inessential, or optional, inputs and outputs. The Validate template is defined to integrate these inputs/outputs into a control structure. For example, it prompts the user in locations where an error message may be written to the report file. Likewise, for disposition of tested records, validate is able to prompt the user at appropriate locations. But the major advantage is the Validate template's knowledge of the types of tests performed in validation and a means of integrating this knowledge with code generation. The user no longer needs the "backtracking" technique of Jackson's methodology to deal with validation tests. The Validate template knows how to handle these tests. There are areas where the Validate template relies on the Jackson methodology, for example, the description of the relationship among the Subject-Record-Source-File, the External-Object-Record-Source-file and/or the Internal-Object-Source when the
latter two are part of the global control structure, i.e., not just accessed local to a single test. In the case of global control structure, the programmer is required to possess an understanding of the structure and the interaction of those files along the lines of the Jackson program structure diagram. He can ignore local test accesses, disposition, and report files which should simplify construction of the program structure diagram.
Chapter 7

Use of the Knowledge in Templates

The previous chapter showed how the problem domain semantics can be represented in a form usable by an automated system, i.e., a software environment. The architecture of such a template-based system is first presented in this chapter with a functional description for each component.

The ability of templates to generate skeletal code was demonstrated in Chapter 6. What was not shown is a scenario that highlights system/user interaction, the interaction of the experts built into the system, and the role of plans in implementing generic tools.

Following the description of the overall system architecture, a scenario for portions of the development process of an application is presented. The scenario uses the "blank forms" concept described by Kuo, Ramanathan, and Li [Kuo82] as a human-engineered means for presenting the grammar details of the template definition. Forms are a common means for collecting information in everyday experience, e.g., tax forms, final and mid-term examinations, university registration forms, etc. Forms present guidance to the
respondent that clarifies the nature of information needed and 
directs the user to blank areas of the form where user specific 
responses are solicited. Menus, a popular type of user interface, 
are a special kind of form. In a menu the choices are first 
presented (user guidance) usually identified by letters or numbers. 
The user then indicates his preference for a choice by responding 
with the corresponding letter or number. 

Blank forms, as defined by Kuo, et al., consist of:

- Headings that describe what is to be filled in.
- Annotations that provide directions on how to fill in the 
  information.
- Entry areas or locations for user-supplied responses.

In an automated system it is possible to tune blank forms and 
the underlying grammar so that they reflect current knowledge of the 
specific application. In the template definitions presented, the 
"T:" local-plan is responsible for utilizing application information 
already collected to accomplish tuning. Further, the "A:" (answer) 
local-plan is responsible for satisfying requests for information 
already supplied by the user or available from other system sources. 
A data dictionary consisting of file descriptions would be used by 
the "A:" local-plan to supply file/database description information, 
thus removing this detail from the user. Where the system is unable 
to supply a specific correct answer, it still may be possible for 
the system to gather pertinent information useful in reaching a
decision. Such choice information is gathered by the "C:" local-plan and placed on the blank form by the "T:" local-plan for the user to choose.

Elements of the scenario were selected to demonstrate system capabilities. Each blank form is followed by the portion of the grammar contributing to the blank form. The actions carried out by the plans are also described, to demonstrate the working of system components.

7.1 Overview of the System Architecture

The template-set expert utilizes global problem domain knowledge contained in the hierarchy of templates to guide the user to the appropriate choice of a template or combination of templates. Once selected, a copy of the individual template is given to the template expert, i.e., local expert. The template expert then is driven by the knowledge represented in that template.

As the system and user interact, a data base of application-specific information is constructed in the form of a derivation tree. The copy of the individual template definition currently utilized by the template expert may be modified by local-plans contained in the template definition to reflect specific application requirements. The architecture as represented in Figure 18 is constructed assuming the existence of a tool-kit as part of the environment. Tool-kits commonly consist of editors, display
managers, database managers, query processors, etc. For a more detailed discussion see [Kuo83a], or [Fish80]. The next section contains the scenario presented as a sequence of subsections. Each consists of the blank form with the completed user responses, portions of the grammar that underlie the blank form, and a discussion of activities.
7.2 Scenario: Interaction with the Template-Set Presentation Expert

The presentation expert contains plans for guiding the user through the hierarchy of templates by

- analyzing the system/user interaction thus far,
- making suggestions to get further information for focusing on the correct template,
- applying appropriate transformations (tuning) to production sets of individual templates.

The application used in the scenario deals with a validate problem. The details of the specification are not important, because the scenario is primarily concerned with showing the nature of the system/user interaction. To accomplish this goal most efficiently the scenario jumps ahead in presenting the interaction, often skipping several blank forms.
7.2.1 Scenario Part A

System Display:

FORM - A1

The data structure environment for your validate problem will be defined first, followed by a definition of relationships among the data structures.

1. Source of records to be tested:

   Enter the file/database name: APRIL-TRANSACTIONS-FILE

   How is the source to be accessed? \textbf{sequentially}

   APRIL-TRANSACTIONS-FILE is implemented as a sequential file. Correct? \textbf{YES} (YES/NO).

   Is the following file definition correct for APRIL-TRANSACTIONS-FILE? \textbf{YES} (YES/NO).

   /* The answer plan located a description under the APRIL-TRANSACTION-FILE name and it is displayed here. */

2. Are the following going to be present in this application?

   External source(s) of test objects \textbf{NO} (YES/NO).

   Internal source(s) of test objects \textbf{YES} (YES/NO).

   Report(s) \textbf{YES} (YES/NO).

   Disposition files for tested nodes \textbf{YES} (YES/NO).
VALIDATE GRAMMAR UNDERLYING FORM - A1

1. validate —>
   data_structure_environment  R: #2
   validate_program_spec       R: #8

2. data_structure_environment —>
   subject_record_source_file_spec  R: #3
   {  external_object_record_source_file_spec  R: #4
      | nil  T: eliminate productions #4, #9, #17
      | nil  T: modify production #8 by removing:
        external_object_record_source_init
        external_object_record_source_final
     }  internal_object_record_source_spec  R: #5
     {  nil  T: eliminate productions #5, #10, #18
        T: modify production #8 by removing:
        internal_object_record_source_init
        internal_object_record_source_final
        IF both external and internal file spec
        are nil
        THEN
        T: eliminate production #48
        T: modify production #34 by removing:
        - local_object_traversal
      ENDIF
     }  report_file_spec  R: #6
     {  nil  T: eliminate productions #6, #11, #19, #43,
         #44, #45, #46, #53, #54
        T: modify productions:
         #8 by removing: report_init, report_final
         #33 by removing: report_test
         #51 by removing: valid_record_report_action
         #55 by removing: invalid_record_report_action
     }  disposition_file_spec  R: #7
     {  nil  T: eliminate productions #7, #12, #20, #38,
         #39, #40, #41, #42, #52
        T: modify productions:
         #8 by removing: disposition_file_init
         disposition_file_final
         #33 by removing: disposition_test
         #51 by removing: disposition_action
         #55 by removing: disposition_process_action
         delay_record_disposition_action
     }
VALIDATE GRAMMAR UNDERLYING FORM - A1 CONTINUED

3. $subject_record_source_file_spec$ →
   $<access_type>(a_t)$
   $<subject_record_source_id>(s_id)$
   $a_t := <access_type>$
   $s_id := <structure_id>$
   A: search data dictionary for description of $s_id$
   IF found THEN
     T: place description found in $<subject_record_source_defn>$ as a choice
     T: place $s_id$ found in $<subject_record_source_type>$ as a choice
   ENDIF

   $<subject_record_source_defn>(s_defn)$
   $s_defn := <subject_record_source_defn>$
   $<subject_record_source_type>(s_type)$
   $s_type := <subject_record_source_type>$
7.2.2 Scenario Part B

Later in the same session, the user is prompted for details about a simple structure that is present.

**System Display:**

---

**FORM - B1**

Are any tests to be conducted on the ATR-SALES-ORDER? **YES** (YES/NO).

If yes, please identify the test type as one of the following:
1) validation test; 2) process test; 3) disposition test; 4) report test. ___.

Are more tests needed for ATR-SALES-ORDER? **NO** (YES/NO)

After all of the tests have been described and before continuing with another structure, enter any additional process code you wish to execute on the ATR-SALES-ORDER.

Choose one of the following next access statements which you have used earlier or write your own.

Selection of next access statements.

1) READ APRIL-TRANSACTIONS-FILE AT END
   MOVE 1 TO EOF-APRIL-TRANSACTION-FILE
   ENDATEND

2) Write your own.

Number of choice to use? ___.
---
VALIDATE GRAMMAR UNDERLYING FORM - B1

15. access_simple_dss →
   <label> (L)
   { test_entry
   | nil
   }
   <process L>(p_l)
   <next_access>(n_a)

33. test_entry →
   { validation_test
   | process_test
   | disposition_test
   | report_test
   }
   { test_entry
   | nil
   }

S: L := <label>
R: #33
T: insert L into the prompt
S: p_l := <process L>
P: p_l
C: collect n_a
T: display n_a
S: n_a := <next_access>
F: n_a
Z: inherit L
R: #34
R: #37
R: #38
R: #43
R: #33
### 7.2.3 Scenario Part C

**System Display:**

<table>
<thead>
<tr>
<th>Form - G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>What trigger type is to be used for this validation test on the Alr-Sales-Order? 4</td>
</tr>
<tr>
<td>Choose one of the following:</td>
</tr>
<tr>
<td>1) Record type determined by record fields.</td>
</tr>
<tr>
<td>2) Record type determined by record position.</td>
</tr>
<tr>
<td>3) Type of data structure.</td>
</tr>
<tr>
<td>4) Previous validation test results.</td>
</tr>
<tr>
<td>5) Execute on all records.</td>
</tr>
<tr>
<td>6) Record field value(s) other than type fields.</td>
</tr>
<tr>
<td>7) Validity of structure.</td>
</tr>
<tr>
<td>8) Other.</td>
</tr>
</tbody>
</table>

If a combination of trigger types is used, enter the numbers from above which describe all the types. _____, _____, _______.

Here are the flags you have defined to remember record and/or structure validity test results. You may want to use them in writing this trigger condition:

- ERROR-ON-COMPLETENESS
- INVALID-ON-TESTS-1-2-3

Please write the "PREVIOUS VALIDATION TEST RESULTS" trigger condition:

- INVALID-ON-TESTS-1-2-3 EQUAL TO "1"

Is the field to be tested (i.e., the subject) by the validity condition in memory? YES (YES/NO)

Is the field to be tested against (i.e., the object) in the validity condition already identified? NO (YES/NO)

Please write any process code you wish to execute prior to evaluation of the validity condition.

- MOVE ATR-SALES-DATE TO WS-DATE
VALIDATE GRAMMAR UNDERLYING FORM - C1

34. validation_test —> 
   test_trigger R: #35 
   { local_subject_traversal R: #47
     | nil 
   } 
   { local_object_traversal R: #48
     | nil 
   } 
   <local_test_preparation_process_action>(l_t_p_p_a)
   S: l_t_p_p_a := <local_test_preparation_process_action>
   P: l_t_p_p_a
   | nil 
   validity_test R: #36
   P: "ENDIF"

35. test_trigger —> 
   S: t_t := null
   P: "IF"

35.1 trigger_type R: #49
   Z: obtain synthesized t_t
   T: insert t_t into prompt 

35.2 <t_t trigger_condition>(t_t_t_cond)
   IF t_t = "validity of structure" OR
   "previous validation test results"
   THEN collect and display r_v_t_r_f attributes
   ENDIF
   S: t_t_t_cond := <t_t trigger_condition>
   P: t_t_t_cond

49. trigger_type —> 
   Z: inherit t_t 
   { record_type_determined_by_record_fields R: none for
     | record_type_determined_by_record_position the entire set
     | type_of_data_structure of alternatives
     | previous_validation_test_results
     | execute_on_all_records
     | record_field_values
     | validity_of_structure
     | other
   } 
   S: s_t_t := nonterminal selected

   { trigger_type(t_t) S: t_t := append(t_t,a_t_t)
     R: #49
   | nil 
   }
7.2.4 Scenario Part D

The following segment continues the scenario and concentrates on the interaction between the template expert and the template-set expert. It is caused by the user's request to locally access a test object from the previous scenario segment.

**System Display:**

**FORM - D1**

Here are the sources of test objects you have defined. Please identify which source you now need to access for the current validity test.  

1) PRODUCT-NUMBER-TABLE  
2) ALLOWABLE-SALES-DATE-TABLE

Here are the local traversals that you previously defined for this source. Select the number of the appropriate traversal.

1) Generate a new traversal.

/* No traversals have yet been defined for this source. */

**VALIDATE GRAMMAR UNDERLYING FORM - D1**

48. local_object_traversal →

C: collect and display all unique values of the e_o_r_s_id (external_object_record_source_id) and the i_o_r_s_id (internal_object_record_source_id) attributes.

C: Collect and display all unique code generated subordinate to the Local_object_traversal nonterminal.

I: If none of the previously generated local_object_traversals are selected, interface with the template-set expert so that the needed traversal can be defined.
7.2.5 Scenario Part E

System Display:

Since ALLOWABLE-SALES-DATE-TABLE is a table to be accessed sequentially, the following templates may be appropriate. Please enter your choice: 

1) H - Traversal of a table for unspecified purpose.
2) F - Linear search of indexed table using SEARCH.
3) G - Linear search of a table with or without indices.
4) P - Binary Search of a table.

ACTIONS UNDERLYING THE TEMPLATE SET EXPERT DISPLAY OF FORM E1

Using the information collected by the "I:" local-plan invoked during the execution of production #48 of the validate template, the template-set expert determines that a table traversal is needed that would access, not initialize, the table. Prompts are customized, and the user is asked to identify the appropriate template from the narrowed selection. Based on the user's choice of 2, the F template will now be used to drive the template expert. Note that the derivation tree constructed using the F template will be a subtree of the partially completed validate derivation tree, and this subtree will be attached as a refinement of the local_object_traversal node.
7.3 Issues Related to the Use of Templates

7.3.1 Template Combinations and Interconnections

The Hypothesis Template Set was designed based on the assumption that data coupling using a sequential, buffered, external file would be adequate to linearly combine templates [Rama82]. This design was simple, it captured the elegant UNIX "pipe" concept, and was motivated by claims such as those made in Constantine -- "Modules must at least pass data or they cannot functionally be part of a single system" [Stev74]. Languages that allow for coroutines are not normally used in this domain, which is why the external file connection mechanism was selected.

During the case study an attempt was made to manually generate a template based solution to the programs being studied. This led to the discovery of several problems with the simple template interconnection mechanism as originally defined. These are described in the following paragraphs. Another observation, that points to the significance of template combination, is that only in a few cases was a single template sufficient to meet the problem requirements and even then it was one of the complex templates that had been defined by combining several lower-level templates. While the template library contains several complex templates that are nested combinations of other templates (e.g., Validate, Update, and Fabricate), the study highlighted the need to also use other
combinations of templates to solve problems.

Templates such as the Case - C and the Series - L provide process branches but have no provisions for traversing external structures. The only way for these two templates to be used is by placing them within another template that supplies the data nodes to be processed.

A second problem with combining templates arises when templates manipulate internal data structures. Templates such as:

- **A** - initialize a table using a fixed value,
- **B** - initialize a table by placing records from a sequential file sequentially into the table,
- **J** - initialize a table from a sequential file by directly placing records into the table,
- **F** - an indexed linear search of a table, and
- **G** - linear search of a table

accept as input or output an internal structure, not an external file. In order to use one of these templates in conjunction with another template, both templates must be able to communicate data through memory instead of an external file. This solution could still be termed data coupling, but it is not through an external file as envisioned in the original Hypothesis Template Set definition.

A third problem with the original connection mechanism occurs when the function \( (F_1) \) accomplished by one template \( (T_1) \) must be synchronized with the function \( (F_2) \) accomplished by another template
(T₂). The point is that F₁ cannot operate on the complete set of nodes before F₂ is initiated and vice versa. The synchronization could be accomplished by designing T₁ and T₂ as co-routines linked by a single node buffer. Unfortunately, languages providing co-routine features are not in common use in the problem domain.

Another manifestation of this problem occurs when the function F₁ of T₁ must be completed during the execution of F₂ of T₂.

The case study identified the need for template combinations facilitated by an external control mechanism(s). The control mechanism should allow for looping through sequences of templates, parallel execution of templates, synchronized execution of templates, and the conditional execution of templates. The control mechanism should maintain the integrity of the individual templates.

Other "non-modular" ways of interconnecting templates also appear necessary. Three possible interactions in this category were identified. These are given below:

1. **Local Embedding**: A template is embedded within the code of another template by preserving the locality of code of the embedded template. The table initialization templates are excellent examples of this category. The embedding is necessary because a template whose focus is a table does not and should not have the complicated input/output interface allowing it to be used in a linear sequence.

2. **Anticipated Non-Local Embedding**: A template is embedded within another without preserving code locality i.e., the code of the two templates will be intermingled. The need to embed is anticipated at template definition time and the mapping of the semantics and productions of the two templates into a newly defined template is done manually.
templates serve as excellent examples of this type of interaction because their semantics are frequently included in the definition of other templates.

3. Unanticipated Non-Local Embedding: Sometimes the need for embedding is discovered only at template instantiation time because of a unique request from the programmer. If the locality of code can be maintained for the embedded template, this category presents no major problems. The difficulty arises when the ad hoc request requires unanticipated migration of code from one template within another, resulting in the intermingling of statements from the two templates. No general mechanism for automating code migration has been found.

Table 18: Limitations on the Interconnection of Templates

1. Templates whose interconnection is limited to embedding only when locality of code can be preserved are: A, B, C, F, G, B, J, P.

2. Templates that can be interconnected using external data structures and whose embedding usually requires code migration - E, I, S, U, V, X, Y.

3. Templates whose interconnection is by way of a linear sequence using external data structures - O.

An interesting question is - "Can nesting of templates be eliminated?" In more abstract terms, consider program P that
consists of statements $x_i$, $y_j$, where $x_i$ is from the X template and $y_j$ is from the Y template. P might then look like Figure 19.

```
x_1
x_2
y_1
y_2
x_3
y_3
y_4
x_4
y_5
```

**Figure 19: Nested Source Lines From Two Templates**

It is assumed that nesting does not alter the sequencing among statements from a template, i.e., $x_i$ precedes $x_{i+1}$. (If the programmer is not bound by logic constraints, nesting can also destroy sequencing within a template.)

Now, can the program represented in Figure 19 always be written as a linear sequence of templates with a simple external file interface like that shown in Figure 20?

The answer, in general, is no, but it depends on the definition of the templates and logic constraints of the problem. For example, if X contains control code that determines when $y_i$ is executed, the solutions available are to nest $y_i$ in the appropriate branch of X's control structure (destroying template modularity) or to pass
control information, e.g., a flag, to Y that controls the execution of $y_i$, thus leading to control not data coupling.

Now, the choices are to either nest templates or increase the degree of coupling between the templates and add control code to Y to evaluate the flags being sent. The relative merits of each must be investigated. For programs consisting of two or more nested templates that cannot be combined by a linear sequence, one of the following approaches might apply:

1. Allow the person defining the template to establish a new composite template that is aware of the statement mingling used and thus can be processed by the environment.

2. Construct a mechanism, expert enough, to allow arbitrarily complex code mingling.

3. Provide the templates and allow the user to define to the system ad hoc code placements. The system will therefore know the identity of the template supplying the statements and thus be able to use this knowledge at maintenance time.

4. Allow more complex coupling. This requires either the existence of global control or requires control code to be added to the receiving template.
One strategy would be to first attempt to use option 1, thus allowing as many common variations of a template as practical, and then allow option 3 to deal with those cases not included in the defined templates. As experience with a template knowledge base increases, the definition of the templates contained within should improve.

Operating systems that allow for concurrent execution of data coupled processes do provide many of the global control mechanisms needed. Where such a system is available option 4 becomes a viable alternative.

The B template, which accomplishes the sequential initialization of a single-level table using the records from a sequential file, requires the use of two separate templates in its definition (i.e., H and I). The H template traverses the table while the I template is used to traverse the file. Here the wish is to investigate the interaction of the two templates assuming each is considered one process in a system of two cooperating concurrent processes. The interface buffer between these two processes will allow room for only one record from the file, thus the two concurrent processes are synchronized. The reason for this restriction is that if a multi-record buffer were provided, the access mechanism added to the second template, namely H, would be as complex as the initial I template. In the nested solution, substituting the concurrent operating system's synchronization
functions (buffer full or empty) for the same function provided by
nesting one template within the other in a single program
(necessarily synchronized in a single processor system) has been
done. The benefit of the use of the concurrent operating system
approach is that modularity of the individual templates is
maintained.

7.3.2 I/O Optimization

A comparison of the original program and the proposed template
solution generated by manually applying the template methodology was
accomplished during the case study. This indicated that in the
template solution the number of I/O operations to secondary storage
often exceeded the I/O operations present in the original program.
The amount could be described by a linear equation of the
information objects present in the inputs and outputs. This is
caused by the need to output records from one template to be then
input to another template.

Traditional programming practice is to write programs that
optimize for execution, often at the expense of program
understandability. Examples of such practices identified during the
case study include: loop reuse, factoring of common adjacent
triggers (see the discussion of the Validate template semantics in
Chapter 4) into a group trigger, and perhaps the most ingrained
- combining processes in order to optimize accesses to secondary
storage. One would like these practices, which adversely affect modularity, to be relegated to an automatic optimizer.
Chapter 8

Conclusions

The software bottleneck is due, in part, to the creative approach currently being taken by software developers in lieu of an engineering approach. Software components which capture programming knowledge and experience in a reusable form are often seen as a necessary component in such an engineering approach. Realizations that commonality exists in programs led practitioners to develop reusable components in several forms. These efforts have not been completely successful because the components defined were not:

- adaptable to the range of requirement variations existing within the problem classes,

- easily combined with others to solve complex problems, and

- complete enough in their knowledge of the problem domain and therefore could only offer superficial guidance to developers.

The fact that commonality in existing programs and problem requirements needed to be identified, defined and organized into a usable form to overcome existing shortcomings, motivated a search for a descriptive, conceptual model of the file/database problem domain. This model served as a foundation for a research methodology and as a vehicle for organizing findings.
The first major contribution of this dissertation has been the development of such a conceptual model. The model developed pointed to the need to divide the problem domain into classes of problems because commonality is present in the semantics of these classes. While some mention of classes of problems was found in the literature, the emphasis was on implementation techniques not on the identification and description of problem class semantics.

Initially, problem classes and their semantics were defined using existing components, programming methodologies, and programming experience with the conceptual model serving as a guiding force. This resulted in the definition of the Hypothesis Template Set.

An extensive case study of production software supplied a wealth of semantics which was identified, defined and categorized using the templates. The case study resulted in definition of the Enhanced Template Set which provided a semantic description of the problem domain, struck a balance between strong semantic content and limited scope of applicability, established the need to use templates in combination, and established the need to provide semi-automatic, human-engineered support for template use.

The identification of semantics, the definition of terminology to describe the semantics and the clustering of the semantics into templates of the Enhanced Template Set has been the second major contribution of this dissertation.
Once defined the semantics of the Enhanced Template Set were used to evaluate the semantic content of the PDM Model Programs and Hypothesized Template Set. Similar use of template semantics holds promise in several areas including: estimation of programming costs, evaluation of software products, and educational applications, such as construction of programming assignments and the rating of educational materials.

Because of the volume of syntactic and semantic detail clustered into individual templates it was clear that a formal mechanism was needed to represent knowledge within templates. The attributed grammar form model, which is a grammar form whose representative grammar is an attributed context-free grammar, provided many of the needed characteristics for encoding template definitions. The grammar form model was useful for describing the instantiation process wherein an application programmer customizes the template to suit specific application requirements by responding to system generated prompts for information. An instantiation grammar from the family of grammars represented by the grammar form was the result of this process. The attributed context-free grammar allowed for the manipulation of software development information that can be of use in both the development and the maintenance processes. Questions were raised as to the ability of the attributed grammar form model to represent all of the tuning operations shown to be needed to make a template-based software
Relationships were shown to exist among template semantics that can be exploited in creating template definitions. A systematic means for producing template definitions by transforming and combining existing template definitions is needed. The attributed grammar form model was less successful in modeling the template definition process. This dissertation has identified practical requirements that are present in the instantiation and definition of templates. It remains as further work to build upon these requirements to produce a new model or continue the extension of the attributed grammar form model began in this dissertation.

"Plans" were defined as procedural elements present in template definitions that extend the semantic functions normally associated with attributed grammars. An architecture of a software environment that uses encoded template definitions and "plans" has been described in the dissertation. The importance of this contribution has been in showing how semantics once identified and represented in a template could be used as the basis for a customizable software environment built upon generic tools.

The ability of templates to address an environment of application variations within a problem class necessarily resulted in complex template definitions. A scenario was presented to demonstrate the interaction of components of the architecture driven by grammar-based template definitions.
8.1 Future Research Directions

Future researchers will no longer have to work with simplified problems for the lack of an understanding of a real, complex, problem domain. The research described in the dissertation has identified the following directions or topics for further study:

1. Search for semi-automatic ways of using problem class semantics to relieve developers of repetitious tasks.

2. Measure the impact of semantics present in the derivation tree constructed using a template-based environment on the maintenance process.

3. Apply the research methodology and representation techniques developed in this dissertation to other problem domains.

4. Implement a software environment which utilizes the templates defined in this dissertation.

5. Find systematic techniques, capable of being automated, for creating template definitions by transforming existing templates.

6. Use these systematic techniques to produce grammar based definitions for the entire Enhanced Template Set.

7. Determine the design of an optimizing tool to overcome the I/O and control code inefficiencies introduced by template combinations and which would replace existing manual attempts at optimization.

8. Identify a formalism which is able to model the requirements of a template-based approach.

9. Apply quantitative measures to the template semantics that allow for accurate estimating of software development costs and completion times.

10. Design a presentation expert capable of responding to user requests for application-specific information contained in the system constructed derivation tree, and
capable of displaying template-driven information requests in an efficient understandable format.
Bibliography


Appendix A.

Human Factors Experiment 1 Materials

The instrument used to administer the first human factors experiment follows.

Experiment 1 - Algorithm Selection

Directions

This packet contains requirements of a problem, an answer sheet, and five abstract flowcharts. After carefully reading the problem requirements you will be asked to consider pairs of flowcharts. All of the flowcharts are logically correct, i.e., they can be used to meet the problem requirements. You will be asked to consider pairs of flowcharts and to state a preference for one flowchart over the other. The questions, i.e., the pairs of flowcharts to be considered as well as an area for recording your preferences are included on the question/answer sheet. Now study the problem requirements, please.

Problem Requirements

This problem deals with the use of a master file to validate the data contained in the transaction file. Data from corresponding transaction records and master records will be used to form an expanded transaction record. Two reports will be generated: an exception report listing any errors discovered and a summary report for valid transactions. What follows is a File Environment Diagram showing the interaction of the files with the program, File Annotations, which describe each of the files, File Record Formats, and a detailed description of the processing required.
A. File Environment Diagram

1. ORDER-LINE-ITEM-FILE - A transaction file referred to as OLI-FILE is sequential, ordered in ascending sequence by Product Number (OLI-PROD-NR), Order Number (OLI-ORDER-NR), and Line Item Number (OLI-LINE-NR). There is 1 record for each line item of an order. See Section B.1 for further details.

2. PRODUCT-MASTER-FILE - A master file referred to as PM-FILE is sequential, ordered in ascending sequence by Product Number (PM-PROD-NR). PM-FILE has 1 record for each product. See Section B.2 for further details.

3. EXPANDED-ORDER-LINE-ITEM-FILE referred to as EOLI-FILE is sequential, ordered in ascending sequence by Product Number (EOLI-PROD-NR), Order Number (EOLI-ORDER-NR) and Line Item Number (EOLI-LINE-NR). See Section B.3 for further details.

4. LINE-ITEM-EXCEPTION-REPORT - referred to as LIER is sequential, ordered in ascending sequence by Product Number (LIER-PROD-NR), Order Number (LIER-ORDER-NR) and Line Item Number (LIER-LINE-NR). Contains 1 record for each invalid transaction. LIER also contains control totals (summaries) ordered by error type, i.e., there are control totals for error type 1 (invalid product number) and separate error totals for error type 2 (invalid unit price).

5. VALID-ORDER-SUMMARY-REPORT - referred to as VOSR contains one record for each group of valid product transactions. The order is ascending by product number.

Annotations:

1. ORDER-LINE-ITEM-FILE - A transaction file referred to as OLI-FILE is sequential, ordered in ascending sequence by Product Number (OLI-PROD-NR), Order Number (OLI-ORDER-NR), and Line Item Number (OLI-LINE-NR). There is 1 record for each line item of an order. See Section B.1 for further details.

2. PRODUCT-MASTER-FILE - A master file referred to as PM-FILE is sequential, ordered in ascending sequence by Product Number (PM-PROD-NR). PM-FILE has 1 record for each product. See Section B.2 for further details.

3. EXPANDED-ORDER-LINE-ITEM-FILE referred to as EOLI-FILE is sequential, ordered in ascending sequence by Product Number (EOLI-PROD-NR), Order Number (EOLI-ORDER-NR) and Line Item Number (EOLI-LINE-NR). See Section B.3 for further details.

4. LINE-ITEM-EXCEPTION-REPORT - referred to as LIER is sequential, ordered in ascending sequence by Product Number (LIER-PROD-NR), Order Number (LIER-ORDER-NR) and Line Item Number (LIER-LINE-NR). Contains 1 record for each invalid transaction. LIER also contains control totals (summaries) ordered by error type, i.e., there are control totals for error type 1 (invalid product number) and separate error totals for error type 2 (invalid unit price).

5. VALID-ORDER-SUMMARY-REPORT - referred to as VOSR contains one record for each group of valid product transactions. The order is ascending by product number.
B.2. Product Master File (PM)

a. Use —
The product master file contains standardized data for each product. It is used to verify the order records contained in the Order Line Item File, i.e., each product ordered must have a corresponding Product Master File record. Also, the unit price must match. The Product Master File is also used to replace the order's item description with a standard description.

b. Record Format —

<table>
<thead>
<tr>
<th>Field</th>
<th>Location</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start</td>
<td>Length</td>
</tr>
<tr>
<td>Status</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Product Number</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Unit Price</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Description</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Create Date</td>
<td>38</td>
<td>6</td>
</tr>
<tr>
<td>Last Update Date</td>
<td>44</td>
<td>6</td>
</tr>
<tr>
<td>Filler</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(All data are characters.)</td>
</tr>
<tr>
<td>Status</td>
<td></td>
<td>Numeric, 0-active, 1-inactive</td>
</tr>
<tr>
<td>Product Number</td>
<td></td>
<td>Numeric</td>
</tr>
<tr>
<td>Unit Price</td>
<td></td>
<td>Numeric -- two decimal places</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Create Date</td>
<td></td>
<td>Numeric - YYMMDD</td>
</tr>
<tr>
<td>Last Update Date</td>
<td></td>
<td>Numeric - YYMMDD</td>
</tr>
<tr>
<td>Filler</td>
<td></td>
<td>Alphanumeric</td>
</tr>
</tbody>
</table>

B.3. Expanded Order Line Item File (EOLI)

a. Use —
The Expanded Order Line Item File contains expanded order records, one for each valid record in the Order Line Item File and two for each invalid record in the Order Line Item File. For every invalid record, the "record type" and "error code" fields are specially set and a second identical record is created, with the RECORD TYPE field set to zero. The EOLI records are similar to the OLI file records except that they contain extra fields, including a standard item description field. The latter data is obtained from the Product Master File record and will be used in the Invoicing Program. (NOTE: The field standard item is nonblank only when the OLI record has a valid product number.) The extended prices are unit price times quantity. When this file is sorted (customer number, order number, line number sequence) in a later program, the invalid records will precede the regular EOLI records for an order and will serve to flag invalid orders.
B. FILE RECORD FORMATS

A number of data files have been prepared for use with the programming problem.

B.1. Order Line Item File (OLI)

a. Use —
   This file contains order data. Each record documents one line item on an order.

b. Record Format —

<table>
<thead>
<tr>
<th>Field</th>
<th>Start</th>
<th>Length</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Type</td>
<td>1</td>
<td>1</td>
<td>Numeric, value = 5</td>
</tr>
<tr>
<td>Customer Number</td>
<td>2</td>
<td>6</td>
<td>Numeric</td>
</tr>
<tr>
<td>Order Number</td>
<td>8</td>
<td>7</td>
<td>Numeric</td>
</tr>
<tr>
<td>Line Number</td>
<td>15</td>
<td>2</td>
<td>Numeric</td>
</tr>
<tr>
<td>Product Number</td>
<td>17</td>
<td>9</td>
<td>Numeric</td>
</tr>
<tr>
<td>Description</td>
<td>26</td>
<td>21</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Quantity</td>
<td>47</td>
<td>3</td>
<td>Numeric</td>
</tr>
<tr>
<td>Unit Price</td>
<td>50</td>
<td>6</td>
<td>Numeric—two decimal places</td>
</tr>
<tr>
<td>Date Ordered</td>
<td>56</td>
<td>6</td>
<td>Numeric</td>
</tr>
<tr>
<td>Salesperson Number</td>
<td>62</td>
<td>3</td>
<td>Numeric</td>
</tr>
<tr>
<td>Filler</td>
<td>65</td>
<td>36</td>
<td>Alphanumeric</td>
</tr>
</tbody>
</table>
### b. Record Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Location</th>
<th>Start</th>
<th>Length</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Type</td>
<td></td>
<td>1</td>
<td>1</td>
<td>Numeric, value = 5: valid record</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: invalid record</td>
</tr>
<tr>
<td>Customer Number</td>
<td></td>
<td>2</td>
<td>6</td>
<td>Numeric</td>
</tr>
<tr>
<td>Order Number</td>
<td></td>
<td>8</td>
<td>7</td>
<td>Numeric</td>
</tr>
<tr>
<td>Line Number</td>
<td></td>
<td>15</td>
<td>2</td>
<td>Numeric</td>
</tr>
<tr>
<td>Product Number</td>
<td></td>
<td>17</td>
<td>9</td>
<td>Numeric</td>
</tr>
<tr>
<td>Description</td>
<td></td>
<td>26</td>
<td>21</td>
<td>Alphanumeric</td>
</tr>
<tr>
<td>Quantity</td>
<td></td>
<td>47</td>
<td>3</td>
<td>Numeric</td>
</tr>
<tr>
<td>Unit Price</td>
<td></td>
<td>50</td>
<td>6</td>
<td>Numeric—two decimal places</td>
</tr>
<tr>
<td>Date Ordered</td>
<td></td>
<td>56</td>
<td>6</td>
<td>Numeric</td>
</tr>
<tr>
<td>Salesperson Number</td>
<td></td>
<td>62</td>
<td>3</td>
<td>Numeric</td>
</tr>
<tr>
<td>Extended Price</td>
<td></td>
<td>65</td>
<td>8</td>
<td>Numeric—two decimal places</td>
</tr>
<tr>
<td>Standard Description</td>
<td></td>
<td>73</td>
<td>21</td>
<td>Alphanumeric—valid record</td>
</tr>
<tr>
<td>Error Code</td>
<td></td>
<td>74</td>
<td>1</td>
<td>Numeric (1-invalid product</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(2-invalid price)</td>
</tr>
<tr>
<td>Filler</td>
<td></td>
<td>95</td>
<td>6</td>
<td>Alphanumeric</td>
</tr>
</tbody>
</table>

#### B.4. Line Item Exception Report

This report should have one line for each Order Line Item Record in error. The error type should be identified. Control totals should be printed by error type. While a single order may contain both errors (i.e., unmatched product number and incorrect unit price) you can only detect and report one error at a time.

#### B.5. Valid Order Summary Report

The Summary Report summarizes order activity by product no. (See format below). It should contain a summary record for each valid product number. Additionally, only the valid order records for a valid product number should contribute to the summary. The reasons for not including invalid records are twofold:

i. They will be processed at a future date and will appear in those sales statistics, and

ii. The appearance of invalid product codes and invalid unit prices in the report would make it suspect.

This report should also contain control totals.
C. Description of the Problem

System Objective

Validate the Order Line Item records using the Product Master file. Use the data from the Order Line Item file and the Product Master file to form an Expanded Order Line Item file. Finally, produce two reports, one reporting exceptions discovered during the validation and the other reporting a summary of the expanded records.

Requirements

1. The following two validation operations are to be performed.
   a. Product Code Validation - An Order Line Item record has a valid product number if there exists a Product Master record with the same product number.
   b. Unit Price Validation - An Order Line Item record has a valid unit price if there is a Product Master record with a matching product number and the unit price in this Product Master record equals the Order Line Item unit price.

An Order Line Item record is invalid if either of the preceding tests fail.

2. The following expansion operations are to be performed for a Valid Order Line Item record.

Create a single Expanded Order Line Item record with record code of 5. This record contains all of the fields from the corresponding Order Line Item record in addition to the following:
   Standard description - from the corresponding Product Master record,
   Error Code - set to 0,
   Extended Price - unit price times quantity.
3. The following expansion operations are to be performed for an Invalid Order Line Item record.

Create two Expanded Order Line Item records for each Invalid Order Line Item record. The first EOLI record with a record code of 5 will be identical to the Expanded Order Line Item record for a valid record except for the following:

- The Standard Description will be spaces if no matching Product Master record was found.
- The Error Code will be set to 1 for an invalid product number or 2 for an invalid unit price.
- The Extended Price will be equal to the Product Master Unit Price times the Order Line Item Quantity unless there is no Matching Product Master record. In which case, the Extended Price will be the Order Line Item Unit Price times the Order Line Item Quantity.

For an invalid record a second Extended Order Line Item record will be produced, this time with a "0" record code and the other fields as described above.

4. The Line Item Exception report will list information about invalid Order Line Item records only.

5. The Valid Order Summary report will list information about Valid Order Line Item records only.

Now, please read the directions on the Experiment 1 Question/Answer Sheet.
Experiment 1 - Algorithm Selection

NUMBERED

FLOWCHART and PSEUDOCODE

PACKET

Each of the solutions is described using a flowchart, as well as the equivalent pseudocode. Only the details of the control code are provided. Data manipulation code has been omitted for simplicity sake.

Flowchart notes

- is a loop symbol
- is an If symbol
- is a generalized I/O symbol
- means to continue at point A
- is the location of point A
FLOWCHART 1

START

READ OLI-FILE and PM-FILE

STOP

A

NOT END-OF-OLI-FILE

F

T

OLI-PROD-NR > PM-PROD-NR

B

C

OLI-PROD-NR = PM-PROD-NR

OLI-PROD-NR = SAV-PROD-NR

F

T

READ OLI-FILE

READ OLI-FILE

READ PM-FILE

READ OLI-FILE
PSEUDOCODE 1

READ OLI-FILE AT END DO
SET END-OF-OLI-FILE-FLAG on
END

READ PM-FILE AT END DO
SET END-OF-PM-FILE-FLAG on
Place the highest possible value in PM-PROD-NR
END

WHILE NOT END-OF-OLI-FILE DO
IF OLI-PROD-NR > PM-PROD-NR THEN DO
READ PM-FILE AT END DO
Place the highest possible value in PM-PROD-NR
END
ELSE DO
IF OLI-PROD-NR = PM-PROD-NR THEN DO
:
:
WHILE OLI-PROD-NR = SAV-PROD-NR DO
:
:
READ OLI-FILE AT END DO
SET END-OF-OLI-FILE-FLAG on
END
ENIWHILE
:
:
ELSE DO
:
:
WHILE OLI-PROD-NR = SAV-PROD-NR DO
:
:
READ OLI-FILE AT END DO
SET END-OF-OLI-FILE on
END
ENIWHILE
:
:
ENDIF
ENIWHILE
STOP
START

READ OLI-FILE and PM-FILE

NOT END-OF-OLI-FILE

STOP

F

OLI-PROD-NR > PM-PROD-NR

T

READ PM-FILE

: : 

READ OLI-FILE

: : 

OLI-PROD-NR = PM-PROD-NR

T

: : 

OLI-PROD-NR = SAV-PROD-NR

F

: : 

READ OLI-FILE

B

: : 

OLI-PROD-NR NOT = SAV-PROD-NR

T

: : 

B

A
FSEUDO'CODE 2

READ OLI-FILE AT END DO
SET END-OF-OLI-FILE-FLAG on
END

READ PM-FILE AT END DO
SET END-OF-PM-FILE-FLAG on
Place the highest possible value in PM-PROD-NR
END

WHILE NOT END-OF-OLI-FILE DO
  IF OLI-PROD-NR > PM-PROD-NR THEN DO
    READ PM-FILE AT END DO
    Place the highest possible value in PM-PROD-NR
  END
  ELSE DO
    IF OLI-PROD-NR = PM-PROD-NR THEN DO
      WHILE OLI-PROD-NR = SAV-PROD-NR DO
        READ OLI-FILE AT END DO
        SET END-OF-OLI-FILE-FLAG on
      END
      ENDWHILE
    ELSE DO
      READ OLI-FILE AT END DO
      SET END-OF-OLI-FILE-FLAG on
      END
    ENDIF
  ENDIF
ELSE DO
  READ OLI-FILE AT END DO
  SET END-OF-OLI-FILE on
ENDIF
ENDWHILE
STOP
FLOWCHART 3

START

READ OLI-FILE
and
PM-FILE

A

STOP

F

NOT ENDE-OF-OLI-FILE

T

OCLI-PROD-NR > PM-PROD-NR

F

OLI-PROD-NR = PM-PROD-NR

T

READ PM-FILE

READ OLI-FILE

OLI-PROD-NR = SAV-PROD-NR

F

T

SAV-PROD-NR = PM-PROD-NR

READ PM-FILE

READ OLI-FILE

OLI-PROD-NR = SAV-PROD-NR

F

T

SAV-PROD-NR = PM-PROD-NR

READ PM-FILE
PSEUDOCODE 3

READ OLI-FILE AT END DO
   SET END-OF-OLI-FILE-FLAG on
END

READ PM-FILE AT END DO
   SET END-OF-PM-FILE-FLAG on
   Place the highest possible value in PM-PROD-NR
END

WHILE NOT END-OF-OLI-FILE DO
   IF OLI-PROD-NR > PM-PROD-NR THEN DO
      READ PM-FILE AT END DO
      Place the highest possible value in PM-PROD-NR
   END
   ELSE DO
     IF OLI-PROD-NR = PM-PROD-NR THEN DO
     ELSE DO
       ENDIF
     END
   END
   READ OLI-FILE AT END DO
   SET END-OF-OLI-FILE on
END
   IF OLI-PROD-NR NOT = SAV-PROD-NR THEN DO
   IF SAV-PROD-NR = PM-PROD-NR THEN DO
   ELSE DO
   ENDIF
   ENDIF
ENDWHILE
STOP
START

READ OLI-FILE and PM-FILE

STOP F

NOT END-OF-OLI-FILE T

OLI-PROD-NR > PM-PROD-NR T

READ PM-FILE

OLI-PROD-NR = PM-PROD-NR T

OLI-PROD-NR = PM-PROD-NR F

READ OLI-FILE

OLI-PROD-NR < PM-PROD-NR T

OLI-PROD-NR < PM-PROD-NR F

READ OLI-FILE

A

B

C
**PSEUDOCODE 4**

READ OLI-FILE AT END DO
SET END-OF-OLI-FILE-FLAG on
END

READ PM-FILE AT END DO
SET END-OF-PM-FILE-FLAG on
Place the highest possible value in PM-PROD-NR
END

WHILE NOT END-OF-OLI-FILE DO
  IF OLI-PROD-NR > PM-PROD-NR THEN DO
    READ PM-FILE AT END DO
    Place the highest possible value in PM-PROD-NR
    END
 ENDIF
  IF OLI-PROD-NR < PM-PROD-NR THEN DO
    WHILE OLI-PROD-NR < PM-PROD-NR DO
      READ OLI-FILE AT END DO
      SET END-OF-OLI-FILE-FLAG on
      Place the highest possible value in OLI-PROD-NR
    ENDWHILE
  ENDIF
  IF OLI-PROD-NR = PM-PROD-NR THEN DO
    : : :
    WHILE OLI-PROD-NR = PM-PROD-NR DO
      : : :
      READ OLI-FILE AT END DO
      SET END-OF-OLI-FILE-FLAG on
      Place the highest possible value in OLI-PROD-NR
    ENDWHILE
  ENDIF
  IF OLI-PROD-NR < PM-PROD-NR THEN DO
    : : :
    WHILE OLI-PROD-NR < PM-PROD-NR DO
      : : :
      READ OLI-FILE AT END DO
      SET END-OF-OLI-FILE-FLAG on
    ENDWHILE
  ENDIF
ENDIF
ENWWHILE
STOP
FLOWCHART 5

START

READ OLI-FILE and PM-FILE

STOP

NOT END-OF-OLI-FILE

OLI-PROD-NR < PM-PROD-NR

OLI-PROD-NR = PM-PROD-NR

OLI-PROD-NR > PM-PROD-NR

READ OLI-FILE

READ PM-FILE

OLI-PROD-NR NOT = SAV-PROD-NR

SAV-PROD-NR = PM-PROD-NR
PSEUDOCODE 5

READ OLI-FILE AT END DO
SET END-OF-OLI-FILE-FLAG on
END

READ PM-FILE AT END DO
SET END-OF-PM-FILE-FLAG on
Place the highest possible value in PM-PROD-NR
END

WHILE NOT END-OF-OLI-FILE DO
IF OLI-PROD-NR < PM-PROD-NR THEN DO
   : 
   :
ENDIF
IF OLI-PROD-NR = PM-PROD-NR THEN DO
   :
   :
ENDIF
IF OLI-PROD-NR > PM-PROD-NR THEN DO
   READ PM-FILE AT END DO
   READ PM-FILE AT END DO
   Place the highest possible value in PM-PROD-NR
END
ELSE DO
   READ OLI-FILE AT END DO
   SET END-OF-OLI-FLAG on
   Place the highest possible value in OLI-PROD-NR
END
IF OLI-PROD-NR NOT = SAV-PROD-NR THEN DO
   IF SAV-PROD-NR = PM-PROD-NR THEN DO
   : 
   :
ELSE DO
   :
   :
ENDIF
ENDIF
ENDIF
ENDWHILE
STOP
Appendix B.

Materials Used Human Factors Experiment Two

Ohio State University
Mr. Shubra

Experiment 2 - Completion of Maintenance Tasks D2RFF1

Introduction

The experiment consists of two activities - Familiarization and Data Collection. During familiarization you will be:

- Introduced to a class of problems that deal with a nested hierarchy of objects.
- Taught a general algorithm to solve such problems.
- Given a specific example of the algorithm dealing with a State University System application.
- Asked to do maintenance on the specific example to assure that you are familiar with the algorithm.

During data collection you will be given two packets each of which contain a problem specification, a maintenance specification, and a program. Taking each packet in turn, you will do the maintenance tasks on the program by interacting with the terminal.

LOGGING IN AND GETTING STARTED

- Login i.e., type "LOGIN CIS788.? 788" and hit RETURN,
- In response to the @ type "PHOTO" and press RETURN,
- Press RETURN again i.e., we will use the default log file name,
- Type "Y" because we want a time stamp,
- In response to the @ type "RUN <CIS788>EXP2DU.EXE",
- Follow the terminal dialog and do what it tells you to do.

Description of the Nested Hierarchy Problem Class

When objects are components of other objects they can be described by a hierarchy. An example of one such application in this problem class is:

A state higher education system composed of universities which contain schools, schools which contain departments, and departments which contain faculty.

This nested hierarchy is shown in the following figure:
<table>
<thead>
<tr>
<th>----- LEVEL -----</th>
<th>----- OBJECT -----</th>
<th>----- STORAGE REPRESENTATION -----</th>
</tr>
</thead>
</table>

Highest level-5  State System  File i.e., all the records taken together.

level-4  University  Groups of schools with the same university id.

level-3  School  Groups of departments with the same university and school ids.

level-2  Department  Groups of faculty records with the same university, school, and department ids.

Lowest level-1  Faculty  A single record.

Nested Hierarchy

Note: Data about a lower level object pertains to the next higher level object. For example, the collection of data about all the departments in a school tells us something about the school. We therefore have a 5 level hierarchy in the State University System example. This example requires 4 key fields in each record. We do not have to identify the state since all the records in the file deal with the same state system.

Since there is only one record for each faculty member, if we want information about the department we must accumulate it by collecting all of the faculty member records having the same department name. To facilitate this collection or accumulation of data about higher level entities files are usually sorted by the key fields with the highest level key being the primary sort key and the lower level keys being taken in order as secondary sort keys. Thus the faculty records belonging to a single department (i.e., having the same department key value) will appear one after the other and faculty members belonging to a school will appear one after the other etc.
One way to design a file to hold a nested hierarchy of objects is to have a fixed record format that contains the data fields for the lowest level object (e.g., faculty), and the key fields of all the higher level objects. If this file design is selected the record format for the example might contain the following fields:
Record Format for the State University System Application

Since the entire file deals with one state system, there is no need to store the State name in every record.

On the next page is a general algorithm that can be used to solve nested hierarchy problems. Take some time to familiarize yourself with the algorithm.

**MAJOR ASPECTS OF THE GENERAL ALGORITHM**

1. A group change is determined by a condition that checks saved keys with the current input key value. These conditions appear in WHILE statements (see statement 80) for the level-2 grouping or IF statements (see statements 210 and 1000).

2. For each level in the hierarchy i.e., at a change in groups, the same processing sequence occurs:
   
   USE level-k data fields
   ACCUMULATE level-k+1 data
   REINITIALIZE level-k data fields
   SAVE level-k keyfields

Where the terms are defined as follows:

INITIALIZING - initial establishing of values
USE - processes that do not combine data items from different levels
ACCUMULATE - lower-level data items are combined to form higher-level data
REINITIALIZE - resetting of data items to initial values
SAVE - placing keyfields from the input record into a save variable so that they are available for comparisons
CONDITION - control code used to traverse the input file
General Algorithm to Solve Nested Hierarchy Problems

10.0 SET END-OF-FILE off
20.0 READ the first record AT END DO
30.0 SET END-OF-FILE on
40.0 END

50.0 SAVE all the key fields of the record just read in
60.0 INITIALIZE all counters and variables used to accumulate information

70.0 WHILE CONDITION for LEVEL-N DO

80.0 WHILE CONDITION for LEVEL-2 DO

90.0 USE the level-1 data i.e., use the record
100.0 ACCUMULATE level-2 data by using the record
110.0 REINITIALIZE for level-1 including:
120.0 READ the next record AT END DO
130.0 SET END-OF-FILE on
140.0 PLACE blanks in the record
150.0 END

160.0 ENDDO

170.0 USE the level-2 data
180.0 ACCUMULATE level-3 data
190.0 REinitialize level-2 counters and accumulators
200.0 SAVE the level-2 input key

210.0 IF CONDITION for LEVEL-3 THEN DO

220.0 USE level-3 data
230.0 ACCUMULATE level-4 data
240.0 REinitialize level-3 counters and accumulators
250.0 SAVE the level-3 input key

260.0 ENDDO

: : : : :

1000.0 IF CONDITION for LEVEL N-1 THEN DO

1010.0 USE level n-1 data
Experiment 2
17 August 1984

1020.0 ACCUMULATE level n data
1030.0 REINITIALIZE level n-1 counters and accumulators
1040.0 SAVE the level n-1 input key
1050.0 ENDF
1060.0 ENDBWHILE
1070.0 USE level n data
1080.0 STOP

The example program on the next page follows the pattern of the general algorithm and meets the following needs for the State University System example:

- a listing of the faculty names,
- a count of the faculty in each department,
- a count of the faculty in each school,
- the total faculty salary in each school,
- a count of the faculty in each university,
- a count of the schools in each university,
- a count of the universities in the state system,
Algorithm to Solve the State University System Problem

10.0 SET END-OF-FILE off
20.0 READ INPUT-RECORD AT END DO
30.0 SET END-OF-FILE on
40.0 END

A
50.0 PLACE IN-UNIVERSITY-ID IN SAVE-UNIVERSITY-ID
51.0 PLACE IN-SCHOOL-ID IN SAVE-SCHOOL-ID
52.0 PLACE IN-DEPT-ID IN SAVE-DEPT-ID

B
60.0 PLACE ZERO IN UNIVERSITY-IN-SYSTEM-COUNT
   FACULTY-IN-UNIVERSITY-COUNT,
   SCHOOL-IN-UNIVERSITY-COUNT,
   SCHOOL-SALARY,
   FACULTY-IN-SCHOOL-COUNT,
   FACULTY-IN-DEPT-COUNT

C
70.0 WHILE NOT END-OF-FILE DO

D
80.0 WHILE IN-UNIVERSITY-ID = SAVE-UNIVERSITY-ID AND
     IN-SCHOOL-ID = SAVE-SCHOOL-ID AND
     IN-DEPT-ID = SAVE-DEPT-ID DO

E
90.0  OUTPUT IN-FACULTY-ID

F
100.0  ADD 1 TO FACULTY-IN-DEPT-COUNT
101.0  ADD IN-FACULTY-SALARY TO SCHOOL-SALARY

G
120.0  READ INPUT-RECORD AT END DO
130.0  SET END-OF-FILE on
140.0  PLACE BLANKS IN INPUT-RECORD
150.0  END

H
160.0  ENDOWHILE

I
170.0  OUTPUT SAVE-DEPT-ID, FACULTY-IN-DEPT-COUNT

J
180.0  ADD FACULTY-IN-DEPT-COUNT TO FACULTY-IN-SCHOOL-COUNT

K
190.0  PLACE ZERO IN FACULTY-IN-DEPT-COUNT

L
200.0  PLACE IN-DEPT-ID IN SAVE-DEPT-ID

M
210.0  IF IN-UNIVERSITY-ID NOT = SAVE-UNIVERSITY-ID OR
      IN-SCHOOL-ID NOT = SAVE-SCHOOL-ID THEN DO

N
220.0  OUTPUT SAVE-SCHOOL-ID, FACULTY-IN-SCHOOL-COUNT,
      SCHOOL-SALARY
230.0 ADD FACULTY-IN-SCHOOL-COUNT TO FACULTY-IN-UNIVERSITY-COUNT
231.0 ADD 1 TO SCHOOL-IN-UNIVERSITY-COUNT
0
240.0 PLACE ZERO IN FACULTY-IN-SCHOOL-COUNT
P
250.0 PLACE IN-SCHOOL-ID IN SAVE-SCHOOL-ID
260.0 ENDF
Q
1000.0 IF IN-UNIVERSITY-ID NOT = SAVE-UNIVERSITY-ID THEN DO
R
1010.0 OUTPUT SAVE-UNIVERSITY-ID, SCHOOL-IN-UNIVERSITY-COUNT, FACULTY-IN-UNIVERSITY-COUNT
S
1020.0 ADD 1 TO UNIVERSITY-IN-SYSTEM-COUNT
T
1030.0 PLACE ZERO IN FACULTY-IN-UNIVERSITY-COUNT, SCHOOL-IN-UNIVERSITY-COUNT
1040.0 PLACE IN-UNIVERSITY-ID IN SAVE-UNIVERSITY-ID
U
1050.0 ENDF
V
1060.0 ENDF
V
1070.0 OUTPUT UNIVERSITY-IN-SYSTEM-COUNT
1080.0 STOP
I want to briefly discuss how the general algorithm can be used to do maintenance on the example program. The general algorithm can act as an index or set of pointers into the details of the application program listing. Given a set of maintenance specifications and actual program statements that can be used to complete the maintenance, you are to first use the general algorithm to determine where the statement would belong in the general algorithm. This will lead to the corresponding location in the application program. You then determine the exact statement number to be used for the given actual program statement. So the process looks like the following:

**GIVEN:**
- Maintenance specification and application program statements
- A pointer into the application program corresponding to the location you selected in the general program

**YOU:**
- Determine where the statement belongs in the general program
- Supply the specific statement number indicating where the program statement belongs

To accomplish the first step you must be able to think at the general level, i.e., to characterize the given program statement in the terms used in the general program. The following terms are used in this general pattern:

- **INITIALIZING** - initial establishing of values
- **USE** - processes that do not combine data items from different levels
- **ACUMULATE** - lower level data items are combined to form data for higher level items
- **REINITIALIZE** - resetting of data items to initial values
- **SAVE** - placing keyfields from the input record into a save variable so that they are available for comparisons
- **CONDITION** - control code used to traverse the input file

Further in this problem you must be able to identify the level of the data involved.

To assure familiarity with the above algorithm, consider the following sample maintenance task:
- For each university we wish to identify the school with the largest number of faculty.

This can be done with a linear search in which BIG-SCHOOL-ID contains the name of the school in a university with the largest number of faculty. BIG-SCHOOL-FACULTY-COUNT contains the count of the largest number of faculty in the school within a university. When the search is complete BIG-SCHOOL-ID will have the school-id we are looking for in the university.

Part of this task is to place the following initializing statements:

PLACE ZERO IN BIG-SCHOOL-FACULTY-COUNT
PLACE BLANKS IN BIG-SCHOOL-ID

Where in the general program do these statements belong or what area in the general program best describes these statements? Think about it!
You should have answered 60, because they are initialization statements. We would then tell you to look at area "B" in the application program. You would then have to enter a specific line number from the application program indicating where each statement should be placed. For example, 60.1 and 60.2.

Now try a few other maintenance tasks.

Assume that the BIG-SCHOOL-ID for a university has been determined. Where would the following statement be placed in the general program?

```plaintext
OUTPUT BIG-SCHOOL-ID
```

Enter your answer here ________

1010 is correct because this is "USE of level-4 data". Level-4 since this is the BIG-SCHOOL-ID for a university i.e., university data. The corresponding location in the application program is "R" and turning to that area in the application program you would give a specific line number for the statement. For example, 1010.1.

Consider the comparison of a FACULTY-IN-SCHOOL-COUNT with the BIG-SCHOOL-FACULTY-COUNT done by the following statement group:

```plaintext
IF BIG-SCHOOL-FACULTY-COUNT < FACULTY-IN-SCHOOL-COUNT THEN DO
  PLACE FACULTY-IN-SCHOOL-COUNT IN BIG-SCHOOL-FACULTY-COUNT
  PLACE SAVE-SCHOOL-ID IN BIG-SCHOOL-ID
ENDIF
```

Where should this statement be placed in the general program? Remember you must be able to categorize the statement group using the terminology of the general program. In this case it is ACCUMULATION of university data i.e., level-4 data. Which is done at statement number 230 in the general program. 230 in the general program corresponds to area "M" in the application program. With that piece of information where would you place the statement group in the application program?

Enter your answer here ________

Raise your hand when you have the answer.

Now type "example" at the terminal and complete the following 2 maintenance tasks. Remember first determine the location in the general program. If you are correct the program will tell you where to look in the application program. The maintenance tasks are:

1. We want to know the salary of each faculty member in the university.

2. We want to know the average faculty age in each department.
Now consider the following problem and maintenance specifications.

**Voter Demographics Application Specification**

Please read over this simplified set of application specifications and the set of maintenance specifications. A file exists that contains voter data. Each record contains data about one voter and the record format is fixed.

<table>
<thead>
<tr>
<th>FIELD NAME</th>
<th>CONTENTS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. VR-REGION</td>
<td>region of the country</td>
<td>e.g., NORTHEAST</td>
</tr>
<tr>
<td>2. VR-STATE</td>
<td>state name</td>
<td>e.g., OHIO</td>
</tr>
<tr>
<td>3. VR-COUNTY</td>
<td>county in the state</td>
<td>e.g., FRANKLIN</td>
</tr>
<tr>
<td>4. VR-PRECINCT</td>
<td>voting precinct</td>
<td>e.g., P-763</td>
</tr>
<tr>
<td>5. VR-VOTER-ID</td>
<td>voter number</td>
<td>e.g., 157-38-8761</td>
</tr>
<tr>
<td>6. VR-VOTER-CONTRIBUTION</td>
<td>contributions made last year to political parties</td>
<td>e.g., $107.85</td>
</tr>
<tr>
<td>7. VR-PARTY</td>
<td>the party the voter is registered with</td>
<td>e.g., Independent</td>
</tr>
</tbody>
</table>

The file is a nested hierarchy of objects with the following nesting:

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>OBJECT</th>
<th>STORAGE REPRESENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Country</td>
<td>Entire file</td>
</tr>
<tr>
<td>4</td>
<td>Region</td>
<td>Group of States with the same VR-REGION</td>
</tr>
<tr>
<td>3</td>
<td>State</td>
<td>Group of Counties with the same VR-REGION and VR-STATE</td>
</tr>
<tr>
<td>*</td>
<td>County</td>
<td>Group of Precincts with the same VR-REGION, VR-STATE, and VR-COUNTY</td>
</tr>
<tr>
<td>2</td>
<td>Precinct</td>
<td>Group of Voters with the same VR-REGION, VR-STATE, VR-COUNTY, and VR-PRECINCT</td>
</tr>
<tr>
<td>1</td>
<td>Voter</td>
<td>Record</td>
</tr>
</tbody>
</table>

* Since the program as initially written does not deal with county as an individual object, the specifications have been written to reflect this.

The file has been sorted into ascending sequence using VR-REGION as the primary sort key with VR-STATE, VR-COUNTY, VR-PRECINCT, and VR-VOTER-ID.
serving as secondary keys in the order listed.

The program as written was to accomplish the following:

- Count the number of voters in each precinct,
- List each voter-id and the party they are registered with,
- List the total voter contributions for each state,
- Produce a count of the number of independents in each state and in the country,
- By region determine the average political contribution by a voter,
- List the total voter contributions for each region.

Maintenance Specification for the Voter Demographics Application

The following maintenance requests for the program have been made:

1. The program is in an infinite loop and appears to be processing the first voter record over and over.
2. After the above problem is fixed you notice that the total voter contributions for the first state are correct, but the second state's contribution total is a sum of the first and second state's contribution.
3. The count of the voters in each precinct is not being displayed.
4. Since the county level was not needed in the original specifications it was not included in the program, but now we want to know the number of counties in the state.

You may refer to this documentation while completing the maintenance tasks. If you wish you can keep track of statement insertions by writing on the program listing. The data collection phase of the experiment is next. As in the example you just finished, you will be asked to first do the maintenance on the GENERAL program by inputting a statement number from the general program that identifies the proper area for the statement. If you correctly identified the area in the general program the terminal will tell you where to look at in the application program. Please locate the document labeled:

GENERAL PROGRAM
It should be on page 12. Also locate the document labeled:
PROGRAM LISTING APPLICATION VOTER
Locate both of these, but

DO NOT

open the coverpage or read the program listings until you enter "BEGIN" and hit RETURN. The terminal will then direct you what to do. As you add (insert) statements you should mark the locations on the listing. Remember, after you type "BEGIN" and hit RETURN data collection starts. Work in an ACCURATE and TIMELY manner. Do not guess at responses!
General Algorithm to Solve 5 Level Nested Hierarchy Problems

1.0 SET END-OF-FILE off
2.0 READ the first record AT END DO
3.0 SET END-OF-FILE on
4.0 END

5.0 SAVE all the key fields of the record just read in
6.0 INITIALIZE all counters and variables used to accumulate information
7.0 LEVEL-5 CONDITION
8.0 WHILE NOT END-OF-FILE DO
9.0 LEVEL-2 CONDITION
10.0 WHILE the level-4 thru level-2 input keys are the same as those saved DO
11.0 USE the level-1 data i.e., use the record
12.0 ACCUMULATE level-2 data by using the record
13.0 REINITIALIZE for level-1 including:
       READ the next record AT END DO
       SET END-OF-FILE on
       PLACE blanks in the record
       END
14.0 ENDWHILE
15.0 USE the level-2 data
16.0 ACCUMULATE level-3 data
17.0 REINITIALIZE level-2 counters and accumulators
18.0 SAVE the level-2 input key
19.0 LEVEL-3 CONDITION
20.0 IF any level-4 thru level-3 input key has changed THEN DO
21.0 USE level-3 data
22.0 ACCUMULATE level-4 data
23.0 REINITIALIZE level-3 counters and accumulators
24.0 SAVE the level-3 input key
25.0 ENDIF
26.0 LEVEL-4 CONDITION
27.0 IF the level-4 input key has changed THEN DO
28.0 USE level-4 data
29.0 ACCUMULATE level-5 data
30.0 REINITIALIZE level-4 counters and accumulators
31.0 SAVE the level-4 input key
32.0 ENDIF
33.0 ENDWHILE
34.0 USE level-5 data
35.0 STOP
Program Listing for the Voter Demographics Application

1.0 SET END-OF-FILE off
2.0 READ VOTER-RECORD AT END DO
3.0 SET END-OF-FILE on
4.0 END

5.0 PLACE VR-REGION IN SAVE-REGION
6.0 PLACE VR-STATE IN SAVE-STATE
7.0 PLACE VR-COUNTY IN SAVE-COUNTY
8.0 PLACE VR-PRECINCT IN SAVE-PRECINCT

9.0 PLACE ZERO IN VOTER-CONTRIBUTIONS-IN-REGION,
COUNTRY-INDEPENDENT-COUNT, STATE-INDEPENDENT-COUNT,
VOTER-CONTRIBUTIONS-IN-STATE, VOTER-IN-STATE-COUNT,
VOTER-IN-PRECINCT-COUNT

10.0 WHILE NOT END-OF-FILE DO

11.0 WHILE VR-REGION = SAVE-REGION AND
VR-STATE = SAVE-STATE AND
VR-COUNTY = SAVE-COUNTY AND
VR-PRECINCT = SAVE-PRECINCT DO

12.0 OUTPUT VR-VOTER-ID, VR-PARTY

13.0 IF VR-PARTY = "INDEPENDENT" THEN DO

14.0 ADD 1 TO STATE-INDEPENDENT-COUNT
15.0 ENDFI

16.0 ADD 1 TO VOTER-IN-PRECINCT-COUNT
17.0 ADD VR-VOTER-CONTRIBUTION TO VOTER-CONTRIBUTIONS-IN-STATE

18.0 ENDFI

19.0 ADD VOTER-IN-PRECINCT-COUNT TO VOTER-IN-STATE-COUNT

20.0 PLACE ZERO IN VOTER-IN-PRECINCT-COUNT

21.0 PLACE VR-PRECINCT IN SAVE-PRECINCT
L
22.0 IF VR-REGION NOT = SAVE-REGION OR
   VR-STATE NOT = SAVE-STATE THEN DO

M
23.0 OUTPUT SAVE-STATE, VOTER-CONTRIBUTIONS-IN-STATE
24.0 OUTPUT STATE-INDEPENDENT-COUNT

N
25.0 ADD VOTER-IN-STATE-COUNT TO VOTER-IN-REGION-COUNT
26.0 ADD STATE-INDEPENDENT-COUNT TO COUNTRY-INDEPENDENT-COUNT
27.0 ADD VOTER-CONTRIBUTION-IN-STATE TO VOTER-CONTRIBUTION-IN-REGION

O
28.0 PLACE ZERO IN STATE-INDEPENDENT-COUNT, VOTER-IN-STATE-COUNT

P
29.0 PLACE VR-STATE IN SAVE-STATE
30.0 ENDIF

Q
31.0 IF VR-REGION NOT = SAVE-REGION THEN DO

R
32.0 AVG-VOTER-CONTRIBUTION = VOTER-CONTRIBUTION-IN-REGION /
   VOTER-IN-REGION-COUNT
33.0 OUTPUT SAVE-REGION, VOTER-CONTRIBUTION-IN-REGION
34.0 OUTPUT AVG-VOTER-CONTRIBUTION

S
T
35.0 PLACE ZERO IN VOTER-IN-REGION-COUNT,
    VOTER-CONTRIBUTION-IN-REGION

U
36.0 PLACE VR-REGION IN SAVE-REGION
37.0 ENDIF
38.0 ENDFHILE

V
39.0 OUTPUT COUNTRY-INDEPENDENT-COUNT
40.0 STOP
Now consider the following problem and maintenance specifications.

**Business Personnel Application Specification**

Please read over this simplified set of application specifications and the set of maintenance specifications which follow.

A file exists that contains business employee data. Each record contains data about one employee and the record format is fixed.

<table>
<thead>
<tr>
<th>FIELD NAME</th>
<th>CONTENTS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER-PLANT</td>
<td>plant id</td>
<td>e.g., Columbus-1</td>
</tr>
<tr>
<td>ER-DEPT</td>
<td>department within the plant</td>
<td>e.g., Computer</td>
</tr>
<tr>
<td>ER-SECTION</td>
<td>section within the department</td>
<td>e.g., Accounting</td>
</tr>
<tr>
<td>ER-PROJECT</td>
<td>project within the section</td>
<td>e.g., AR-115-C</td>
</tr>
<tr>
<td>ER-EMPLOYEE-ID</td>
<td>employee id</td>
<td>e.g., 157-67-7890</td>
</tr>
<tr>
<td>ER-EMPLOYEE-SALARY</td>
<td>employee’s salary</td>
<td>e.g., $24100.00</td>
</tr>
<tr>
<td>ER-EMPLOYEE-DEGREE</td>
<td>employee’s highest degree</td>
<td>e.g., PhD</td>
</tr>
</tbody>
</table>

The file is a nested hierarchy of objects with the following nesting:

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>OBJECT</th>
<th>STORAGE REPRESENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Company</td>
<td>Entire file</td>
</tr>
<tr>
<td>4</td>
<td>Plant</td>
<td>Group of Departments with the same ER-PLANT</td>
</tr>
<tr>
<td>3</td>
<td>Department</td>
<td>Group of Sections with the same ER-PLANT, and ER-DEPT</td>
</tr>
<tr>
<td>*</td>
<td>Section</td>
<td>Group of Projects with the same ER-PLANT, ER-DEPT, ER-SECTION</td>
</tr>
<tr>
<td>2</td>
<td>Project</td>
<td>Group of Employees with the same ER-PLANT, ER-DEPT, ER-SECTION, and ER-PROJECT</td>
</tr>
<tr>
<td>1</td>
<td>Employee</td>
<td>Record</td>
</tr>
</tbody>
</table>

* Since the program as initially written does not deal with section as an individual level, we have not included it as a separate level.

The file is in ascending sequence by ER-PLANT, ER-DEPT, ER-SECTION, ER-PROJECT, and ER-EMPLOYEE-ID primary sort key to lowest level sort key respectively.

The Program as written was to accomplish the following:

- List each employee id and their highest degree,
Experiment 2 17 August 1984 Page 19

- Count the number of employees working on each project,
- List the total salary for each department, i.e., the sum of all an
department's employee salaries,
- List the total salary for each plant,
- Produce a count of the number of PHD's in each department and in the
company,
- By plant determine the employee's average salary.

A note on pseudocode:
EXECUTE PROC-A THRU PROC-X
WOULD CAUSE TRANSFER OF CONTROL TO PROC-A AND WOULD EXECUTE ALL THE
PROCEDURES UPTO AND INCLUDING PROC-X, RETURNING CONTROL AT THE END OF PROC-X.
EXECUTE PROC-A
WOULD EXECUTE PROC-A ONLY AND THEN RETURN.

Maintenance Specification for the Business Personnel Application

The following maintenance requests for the program have been made:

1. The program is in an infinite loop and appears to be processing
the first employee record over and over.
2. Having fixed the above problem you notice that the total salary for
the first department is correct, but the second department's
salary is a sum of the first and second departments' salaries.
3. The count of the employees working on each project is not being
displayed.
4. Since the section level was not needed in the original
specifications it was not included in the program, but now we want
to know the number of sections in each department.

You may refer to this documentation while completing the maintenance tasks.
The data collection phase of the experiment is next. The next page of this
document is labeled:

PROGRAM LISTING APPLICATION
EMPLOYEE
Locate it, but
DO NOT
open the coverpage. Before you look at the program listing enter "BEGIN" and
hit RETURN. The terminal will then direct you what to do. As you add
(insert) statements you should mark the locations on the listing. Remember,
after you type "BEGIN" and hit RETURN, data collection starts. Work in an
ACCURATE and TIMELY manner. Do not guess at responses!
Program Listing for the Business Personnel Application

1.0 SET END-OF-FILE off
2.0 READ EMPLOYEE-RECORD AT END DO
3.0 SET END-OF-FILE on
4.0 END

5.0 PLACE ER-PLANT IN SAVE-PLANT
6.0 PLACE ER-DEPT IN SAVE-DEPT
7.0 PLACE ER-SECTION IN SAVE-SECTION
8.0 PLACE ER-PROJECT IN SAVE-PROJECT

9.0 PLACE ZERO IN PLANT-IN-COMPANY-SALARY,
   COMPANY-PHD-COUNT, DEPT-PHD-COUNT,
   DEPT-IN-PLANT-SALARY, EMPLOYEE-IN-DEPT-COUNT,
   EMPLOYEE-ON-PROJECT-COUNT

10.0 WHILE NOT END-OF-FILE DO
11.0 IF ER-PLANT NOT = SAVE-PLANT THEN DO
12.0 EXECUTE PROJECT thru PLANT
13.0 ELSE
14.0 IF ER-DEPT NOT = SAVE-DEPT THEN DO
15.0 EXECUTE PROJECT thru DEPT
16.0 ELSE
17.0 IF ER-SECTION NOT = SAVE-SECTION OR
18.0 ER-PROJECT NOT = SAVE-PROJECT THEN DO
19.0 EXECUTE PROJECT

20.0 ENDCASE
21.0 IF ER-EMPLOYEE-DEGREE = 'PHD' THEN DO
22.0 ADD 1 TO DEPT-PHD-COUNT
23.0 ENDIF
24.0 ADD ER-EMPLOYEE-SALARY TO DEPT-IN-PLANT-SALARY
25.0 ENDWHILE
26.0 OUTPUT COMPANY-PHD-COUNT
27.0 STOP
28.0 PROJECT
29.0 ADD EMPLOYEE-ON-PROJECT-COUNT TO EMPLOYEE-IN-DEPT-COUNT
30.0 PLACE ZERO IN EMPLOYEE-ON-PROJECT-COUNT
31.0 PLACE ER-PROJECT IN SAVE-PROJECT
32.0 DEPT
33.0 OUTPUT SAVE-DEPT, DEPT-IN-PLANT-SALARY
34.0 OUTPUT DEPT-PHD-COUNT
35.0 ADD DEPT-IN-PLANT-SALARY TO PLANT-IN-COMPANY-SALARY
36.0 ADD DEPT-PHD-COUNT TO COMPANY-PHD-COUNT
37.0 ADD EMPLOYEE-IN-DEPT-COUNT TO EMPLOYEE-IN-PLANT-COUNT
38.0 PLACE ZERO IN DEPT-PHD-COUNT, EMPLOYEE-IN-DEPT-COUNT,
39.0 PLANT
40.0 PLACE ER-DEPT IN SAVE-DEPT
41.0 PLANT
42.0 AVG-EMPLOYEE-SALARY = PLANT-IN-COMPANY-SALARY /
   EMPLOYEE-IN-PLANT-COUNT
43.0 OUTPUT SAVE-PLANT, PLANT-IN-COMPANY-SALARY
44.0 OUTPUT AVG-EMPLOYEE-SALARY
45.0 PLACE ZERO IN PLANT-IN-COMPANY-SALARY, EMPLOYEE-IN-PLANT-COUNT
46.0 PLACE ER-PLANT IN SAVE-PLANT
Appendix C.

Template Descriptions and Summary Data

Resulting from the Case Study of Production Programs

This appendix identifies critical terminology used by programmers during the design and implementation of software for the file/database problem domain. The terminology was identified and defined during a series of efforts which culminated in a case study of 40 industrial programs. This appendix documents the data obtained during the case study and describes the implications of the data with respect to defining different templates.
C.1 Generalized Traversal Template - I

WHEN IS THIS TEMPLATE APPLICABLE?

The primary source of complexity in the file/database processing domain is the generation of control code to traverse structures containing data. Except for two of the primitive templates (Case and Series) which must be nested in other templates to be useful, the Generalized Traversal - I template semantics are incorporated in the definition of every other template. Control code can be generated using this template to traverse any multiple file relationship capable of being expressed as a Jackson program structure diagram [Jack75]. Other than collecting source identity and access information the traversal of a multiple source program structure diagram reduces to problems similar to the traversal of a single source and is therefore discussed in those terms in the remainder of this chapter.

Applications dealing with a nested hierarchy of objects are commonplace in the file/database problem domain. The nested hierarchy is a collection of 1:N relationships between an object at level i and many objects at level i+1. Traditional sequential files ordered by several sort keys contain such a nested hierarchy, as does the "SET" construct of the Codasyl DBMS or the tree structure of IBM's IMS database system.
C.1.1 Template Semantics and Related Case Study Observations

C.1.1.1 Traversal of the Hierarchy of Objects

THE RELATIONSHIP BETWEEN SORT KEYS AND LEVELS IN THE HIERARCHY

To construct an algorithm to traverse a nested hierarchy, the programmer must know how the hierarchy was implemented (i.e., sequential file, Codasyl DBMS or hierarchical DBMS); how may levels occur in the hierarchy; the number, type and sequence of objects appearing on each level; and the criteria used to identify related records. By far, the most common implementation of the nested hierarchy is the sorted sequential file containing a single record type. Each record includes the data for one occurrence of the lowest-level object in the hierarchy along with the key fields for all the objects in its hierarchical path. Three sort keys could be used to obtain the hierarchies in Figure 21 or Figure 22.

The difference between Figures 21 and 22 is that the lowest-level object, OBJ4 in Figure 22, either does not have a key field or the records need not be ordered on the OBJ4 key field. While the lowest-level object in Figure 21 will be ordered, no additional information object, or grouping of records is created by using Sort Key 3. Figure 21 therefore, provides four information entities using three sort keys. Three of the entities are realized by partitioning records. See Table 19.

The final information object (OBJ3) in Figure 21 is represented
by a single record. Sort Key 3 in Figure 21 is not used to partition records, but merely to order OBJ3's.

Figure 22, uses 3 sort keys also, but provides five information entities, four of which are realized by partitioning. The
Table 19: Partitions Present in Figure 21

<table>
<thead>
<tr>
<th>Information Entity</th>
<th>Partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>All records</td>
</tr>
<tr>
<td>OBJ1</td>
<td>Records with equal sort key 1</td>
</tr>
<tr>
<td>OBJ2</td>
<td>Records with equal sort key 1 and sort key 2</td>
</tr>
</tbody>
</table>

partitions include the three in Table 19 plus OBJ3 which is a grouping of records with all three sort keys equal. In Figure 22, the fifth information entity, OBJ4, is represented by a single record. A sort that uses the key field of the lowest-level object, assuming that the lowest-level object is represented by a single record occurrence, does not create an additional grouping, but merely orders records that are already grouped.

VIRTUAL OBJECTS

Objects represented by collections of other objects will be called "virtual" objects because the data about those objects is not represented by a record type, but must be manufactured by the processing of lower-level objects.

Structures present in Figures 21 and 22 can be traversed using a nested loop algorithm. Figure 21 requires 3 nested loops while Figure 22 requires four nested loops. The three nested loops for Figure 21 serve the following purposes – the inner most loop
processes all of the OBJ3's belonging to a single OBJ2, the middle
loop processes all of the OBJ2's belonging to a single OBJ1, and
finally, the outer loop process all of the OBJ1's belonging to the
file. No loop is needed to process an instance of OBJ3, since all
of the data about a single occurrence of an OBJ3 is available in a
single record.

Each level of the nested loop implementation, contains
locations to first initialize for the level, then accumulate data
for the level by processing all of the next lower-level objects
belonging to this level, and finally use the data generated for the
current level. Initialization includes resetting counters and
accumulators, along with saving and perhaps forming the key value
for identifying the group. Looping through the lower-level objects
allows counting and accumulating of the data needed at higher
levels. The use of the data at a level takes two forms: the data
can be output, and/or the data can be used to form data for the next
higher-level object.

**SKIPPING AN INTERMEDIATE-LEVEL VIRTUAL OBJECT**

A variation of the standard control break occurs when an
intermediate-level is skipped. Although it is present (at least
identified by a key field), it is being ignored in the current
application. Consider Figure 23.

If the application ignores OBJ2, and a sequential file
implementation is used, the hierarchy becomes that shown in Figure 24.

Assuming SORT Key 3 does not uniquely identify OBJ3, SORT Key 2 is still required to differentiate between OBJ3's. Note, only 3
nested loops, not the original 4, are needed in the simplified application.

Consider the Bachman diagram in Figure 25 that represents a Codasyl DBMS implementation of the nested hierarchy problem.

```
SYSTEM
    | S-01 SET
    |
OBJ1
    | O1-02 SET
    |
OBJ2
    | O2-03 SET
    |
OBJ3
    | O3-04 SET
    |
OBJ4
```

**Figure 25:** Codasyl DBMS Implementation of a Nested Hierarchy

While this is a typical database design for the nested hierarchy problem, it does differ in several significant ways from the typical sequential file design. If you recall, the sequential
file had one record type that was used to represent the lowest-level object (OBJ4). The other objects (File through OBJ3) were all virtual. In the CODASYL design, 5 record types are present. Each of these types can be used to store data about the object; therefore, completely virtual objects are no longer present. Usually, not all of the data about an object is available by accessing a record. Some of the data must still be accumulated by processing lower-level objects.

Now, what is the effect of ignoring a level in the Codasyl implementation? If OBJ2 is again ignored because it is not important to the current application, OBJ2 key cannot simply be concatenated with OBJ3 key and forgotten, as was done in the sequential file implementation. While a subschema can be generated to shield the fields present in OBJ2 from the program, the 01-02 set and 02-03 set will still have to be traversed to process OBJ3's. The problem is that the pointers that link OBJ1 with OBJ3 are stored in OBJ2 records. Further, although key fields may be duplicated in subordinate nodes, this is not the normal practice. So, Key 2 must be gleaned from the OBJ2 record. Therefore, four nested loops are needed to traverse the structure even though an intermediate level is being ignored.

In an IMS implementation of the hierarchy with OBJ2 ignored, the segment search arguments, commonly called "SSA's," can be written so as to skip over any OBJ2 segments. Key 2 will still be
available in the program communication area as part of the composite key which is a concatenation of all the key fields in the hierarchical path of a segment. An IMS implementation would require only three nested loops as did the sequential file implementation.

C.1.1.2 The Criteria for Grouping Key-Related Records

In dealing with record grouping defined by keys, the values of fields present in the current record are compared with the fields defining the current group to determine if the current record belongs to the current group or not. If related records (those belonging to a group) are adjacent in the input structure (either by juxtaposition or a pointer list), the discovery of a current record that does not belong signals the end of the current group and the beginning of the next group. At the end of a group, the accumulated data is used, and set up for the new group which includes reinitializing and establishing the values identifying the new group is accomplished. There are two common ways of establishing the controlling values for the new control group. By far the most common is to use the key values of the current record to define the keys of the new control group, but other methods are possible. An outside structure such as a table or file may contain the group definition values. For example, if daily sales records are to be grouped into weekly totals, the ending or starting date of each week will have to be defined. Sales record dates would then be compared
with this group date. Assuming that the sales file is sorted in ascending sequence by date, as soon as a sales date exceeds the end date of the current group, a new group is begun. If the group test values are not taken from the current record, then the algorithm must allow for defining the new group test values.

C.1.1.3 The Effect of Multiple Record Types on File Traversal

What is the effect on the traversal algorithm of having several different record types on a single level of a hierarchy? In terms of Jackson diagrams, this situation is represented by one of the two diagrams in Figure 26.

The situation represented in Figure 26 part 1 can be traversed in two possible ways. See Figures 27 and 28. Both methods assume a record is in place prior to reaching the level (this is the result of a priming read), and the level will be exited with a record belonging to another level or group in place and not yet processed.

While both traversals will process structures that correspond to Figure 26, part 1, Figure 27 conforms more closely to the constraints present in Figure 26 than the alternate traversal does. Figure 28 requires a loop as a part of the next higher level in order to process the known sequence of N records. The Figure 28 traversal, which assumes the existence of a higher-level loop, will process objects appearing in any order (not the strict order depicted in the diagram) and any number of objects of each type may
1. Known sequence of record types.

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>record</td>
</tr>
<tr>
<td>record</td>
</tr>
<tr>
<td>record</td>
</tr>
</tbody>
</table>
```

2. Mutually exclusive collection of possible record types.

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>record</td>
</tr>
<tr>
<td>record</td>
</tr>
<tr>
<td>record</td>
</tr>
</tbody>
</table>
```

**Figure 26:** Two Structures Representing Multiple Record Types on a Single Level of a Hierarchy

The structure represented in Figure 26, part 2 calls for an n-level case structure. Again, assuming a record already in place and a new record accessed before exiting the level, the traversal looks exactly like Figure 28. But, using the traversal present in Figure 28 for a mutually exclusive collection of possible record types does not require a loop at the next higher level, as would be the case if
the Figure 28 traversal is used for a known sequence of record types.

C.1.1.4 Generation of Access Statements for the Nested Hierarchy

The generation of file access statements is a complex issue. As a starting point, let's assume a priming read algorithm is to be used in traversing the sequential data source (file or database) as shown in Figure 29. This requires a priming read with a second read embedded in the looping control structure.

The major concepts that control the generation of additional read statements to traverse a nested hierarchy are:
IF <object 1>
   <process object 1>
ELSE
   IF <object 2>
      <process object 2>
   ELSE
   ENDIF
   <access next>
ENDIF

Figure 28: Alternate Traversal of the Structure Represented in Figure 26, Part 1

1. Record types - a unit of I/O access that is differentiated from other access units by either position (first in the file) or the value of a field(s).

2. Virtual Objects - data pertaining to virtual objects are not available in a single record but must be accumulated by processing subordinate objects.

3. Partitioning of records - grouping together records that have common characteristics (e.g., equal key fields or a field within a specified range; or members of the same set occurrence.)

A source structure usually contains a small number of predefined record types (most often a single type per sequential
Code | Purpose
--- | ---
READ | 1. Priming read of any type record
DO WHILE
... READ | 2. Read next
... ENDDO

Figure 29: Priming-Read Algorithm

file). Multiple record formats, or multiple levels in a DBMS model are a good indication of the existence of different types of records, where each record format is a type and each DBMS level contains 1 or more types. It is possible for multiple record types to exist where all types have the same record format. A field(s) in the record is usually designated to identify the type of the record. While this practice is common with traditional access methods, it is less common with database systems. If a record does not have a type identification field as part of the record, then either the system must indicate the record type to the program (this is the case in most database software) or the context of the record must identify the type. Position is frequently employed to define file
header records, i.e., the first record read has file summary or control data. There are two possible effects of multiple record types on the generation of access statements. Either the programmer knows the sequence of types and takes advantage of this knowledge by generating additional process code followed by a Read (see Figure 27), or a control structure (like Case) is established to ascertain the type of the record (see Figure 28).

The data for virtual objects must be accumulated or built by processing subordinate objects. The file itself is an excellent example of a virtual object which requires a summary program to produce information about the entire file. Although virtual objects require additional process statements to use accumulated data, they do not require any Read statements beyond those present in the priming-read algorithm.

Grouping together records with a common set of characteristics (e.g., equal key value) for the purpose of processing the records in the group is called partitioning. Three major implementation techniques are used to effect partitioning within a single source (database or file). These implementation techniques are:

1. Key groups where each of the records, no matter what the type, contains common key values. The grouping can then be accomplished by using the common key fields as sort keys.

2. Delimiter groups, where records are preceded and perhaps followed by special partitioning record types (delimiter records). The records in the group do not have to carry any physical identification of the partition other than
being located within the delimiters.

3. Partition maintained by system software such as a DBMS package. The Codasyl "Set" or the "Parent" in IMS are examples of such a partitioning mechanism.

Key groups have no effect on the access statements generated other than what would be generated by additional record types within the group.

Delimiter groups by definition contain at least two record types -- the delimiter type and type of the records being grouped by the delimiter. Recalling the discussion on record types, delimiter groups would require an additional PROCESS and READ (for the begin-partition delimiter) prior to looping through the records in the partition and an additional PROCESS and READ following the loop if an end-partition delimiter is employed. Figure 30 shows the priming read algorithm for a single level delimiter group with leading and training delimiters.

The Read on line 1 is the original priming read and the Read on line 8 is the original nested read, both from the priming-read algorithm. The Read on line 5 serves as a priming read for records in the partition although, if the partition is empty, it will read the trailing delimiter for this empty group. The Read on line 11 is the priming read for the leading delimiter of the next partition. The Jackson diagram for such a structure is shown in Figure 31.

When dealing with a system-maintained partition like Codasyl Sets, each set has at least two types, an owner type and a member
type. Zero to N occurrences of the member type may be present. Sequential entry to a set occurrence may be through its owner or through a member within the set. Owner entry occurs when a record is a member in one set type and the owner in a subordinate set type. Member entry to a set occurs when a record is a member in two different set types. The two situations are depicted in the following Bachman diagrams (Figure 32 and Figure 33) where the B-type record is used to gain entry to a set in both cases.

Now consider these two possibilities and their effect on the generation of access statements (see Figures 34 and 35). In both cases, assume that the B-type record has already been accessed. See Figure 34. The Read on line 1 serves to access the first C-type record (priming read for SET BC members). The read on line 4 will
only access C-type records in the current BC set. Upon exhausting
**Figure 33:** Set CB Entry Through Its Member B

```
1 READ
2 DO WHILE <not end of set BC>
3   <process>
4     READ
5 ENDDO
6 <process>
7 READ
   . . .
```

**Figure 34:** Set BC Traversal Using Owner Entry

This occurrence of Set BC, a set empty status will be sent instead of reading the next B-type record. The read on line 7, therefore,
serves to access the next B-type record.

While processing SET AB with an A-type and B-type record already accessed, the programmer might wish to traverse SET CB. The code with access statements that accomplishes this kind of traversal is presented in Figure 35.

```
1       READ
2       <process>
3       READ
4       DO WHILE <not end of set CB>
5            <process>
6            READ
7       ENDDO
8       <process>
9       READ
          ...
```

*Figure 35: Set CB Traversal Using Member Entry*

The read on line 1 accesses the owner of the current set CB as identified by the current B-type member record. The read on line 3 is now the priming read for the traversal of the CB set. The read on line 6 will only access B-type records belonging to the current CB set and return an end of set status when no more records exist. The read on line 9 is, therefore, needed to access the next B-type record in the current AB set.
Overview of Programs Using the Generalized Traversal Template

1. ACH400 — A case template is used to control which of 2 possible validation templates are executed. The program selects records for processing and deals with logical record groups defined by header and trailer delimiters. The traversal used in the disk version is reported here.

2. IMS2 — A program that accesses a database and produces a specialized report (multiple report columns) from source 1, and fabricates transaction records to be used in updating two other databases. Upon reaching the end of source 1 a second source is processed. Selected records are used to update, an IMS database. The database is directly accessed using segment search arguments to locate the proper position. The traversals of the first and second sources are both reported.

3. RD065 — Records from source 1 are used to fabricate records which are then sorted. The sort forms a 2-level hierarchy in source 2 that is used to fabricate the final output file.

4. VDC65Q — Under parameterized control (the parameter identifies key groups to be processed) a key group is selected and used to fabricate records for an output file.

5. VDM602 — Selected records from a database are used to fabricate records output to a posting file. The database records are then deleted.
C.1.3 Summary Data for Programs Using the Generalized Traversal Template

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

1. Total Nr. of Sources

| Program | 1 | 2 | 2 | 2 | 1 |

2. Sources by Type

<table>
<thead>
<tr>
<th>Source</th>
<th>File</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

3. By Source Nr. of Rec Types

<table>
<thead>
<tr>
<th>Program</th>
<th>S1</th>
<th>S2</th>
<th>S1</th>
<th>S2</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec Types</td>
<td>5</td>
<td>17</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Group Types</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Key</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

4. Deepest Group Nesting

| Program | 1 | 1|0 | 2|2 | 2|0 | 3 |

5. Max. Fixed Seq. Length

| Program | 3 | 4|N/A | 2|1 | 3|1 | 2 |

6. Max Variable Seq. Length

| Program | N/A | N/A|2 | N/A | 2|N/A | 7 |

7. Selectivity by

<table>
<thead>
<tr>
<th>Selectivity</th>
<th>All</th>
<th>Y</th>
<th></th>
<th>Y</th>
<th></th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec. Type</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
C.2 The Fabricate Template - Y

WHEN IS THE TEMPLATE APPLICABLE?

The Fabricate template accepts data from one or more sources, manipulates the data and produces one or more records with the desired format and content. Reports for a variety of purposes are also frequently generated. The template discussion and data collection is organized around Inputs, Outputs, their relationships, and the kinds of data manipulations that were observed in the case study.

Fabricate can be applied to overcome physical storage restrictions such as the limitation of 80 column cards. For example, a program from the case study sample (RECON510) had three-card record groups with each card in a group containing identical key values. These three-card groups were reformatted into a single large disk record thus simplifying future program traversals (a single access versus three) and conserving disk space by the elimination of duplicate fields. The Fabricate template may be used to simplify other programs by transforming/grouping input fields into the form needed by the other programs. Quite often the Fabricate template is used as the input procedure for a Cobol sort verb. Its use here is to eliminate unneeded record fields thus shortening the size of the record to be sorted, or to place key fields from several record types into fixed locations which are then
used as sort keys. A final motivation for fabrication may be to remove or transform user-friendly mechanisms (e.g., blank numeric fields or display usage) into efficient processing and storage forms (e.g., zeroes in the numeric fields or packed decimal usage).

C.2.1 Template Semantics and Related Case Study Observations

C.2.1.1 Input Data

Source data is either scattered over many records in a single file/database/table or over many records/nodes residing in separate files/databases/tables. Fabricate, through the use of the Generalized Traversal template semantics provides the traversals needed to bring related records together. Records can be selected by record type, key value, position in the file, or by procedure (e.g., using validity checks or other such complex procedures). Selected records are processed further by Fabricate while unselected records are ignored. When reporting the case study results, the structure of each source in terms of record types and logical grouping is defined. Relationships among sources where several sources are present are also reported. These source to source relationships could be 1:1, 1:M, M:1 or M:N.
C.2.1.2 Output Data

Two categories of outputs were observed: the files/databases/tables that contained the fabricated records, and the reports that chronicled the fabrication process. Like the inputs, the structure of fabricated outputs is reported by record types and logical grouping. Fabricated records need not all be sent to a single output. Often they are partitioned into several output files. Where such partitioning exists the partitioning criteria was recorded. The relationship between each input and each fabricated output was also identified and reported.

Reports generated during fabrication provided:

- a list of any error situations;
- details or summaries of inputs,
- details or summaries of outputs,
- an audit trail showing the relationship between inputs and generated outputs, and
- a listing of check sums computed from inputs and outputs.

C.2.1.3 Fabrication Actions

Once appropriate input records are present in memory, several kinds of actions can occur:

- As a result of tests, it can be determined that the records are not to be further processed.
- The grouped input records could be reported but would not contribute to the creation of any new records.
- The records could be copied, intact, to the fabricated output.

- The records could participate in a data manipulation action(s) resulting in a fabricated record.

The criteria used to determine the fabricate action to be executed include:

- criteria that is fixed, i.e., each record or record group is subjected to the same action(s), or

- criteria that is dependent on record type, record key or key group, position in the input, validity, or a detailed procedure.

While the input and output semantics of Fabricate overlap with other templates such as Series and Report, they are distinguished by the nature and the richness of possible data manipulation actions. The Fabricate data manipulation actions could require:

- copying of all fields of selected records,

- reformatting of source fields by reordering all fields, or reordering selected fields where the selection can be fixed or dynamic,

- encoding of a field which means substituting a cryptic value for a longer more descriptive value,

- decoding of a field which is the inverse of encoding,

- accumulating of fields from several sources either maintaining field integrity or combining field values in such a way (e.g., addition) that the initial values are no longer distinguishable,

- combining of fields (perhaps from several records) from one source again either maintaining field integrity or losing individual field identity,

- expanding a single source record into two or more
fabricated records (e.g., creating a record for each item in a repeating group),

including fields (such as sequence numbers, dates, record tags) maintained by the program.
C.2.2 Overview of Programs Using the Fabricate Template

1. DQ010 — Records are first validated with valid records being used to fabricate an output file. Because the dependent status field is reformatted, Validate record disposition is not sufficient and Fabricate is used. Also, the program concurrently generates detail and total lines for three reports — error, fabricated record report, and check sum audit trail. Normal report semantics (titles, page breaks) are not used during the generation of the detail and total lines. Therefore, Fabricate could have been selected to represent the formation of the report lines. Closer inspection shows that the error and check sum audit trail reports are allowed in the Validate template and the fabricated record report can be generated during the creation of the output transactions file.

2. D140 — Twenty-four record types are present, each containing a validity flag which is set in earlier programs. Selected records (by record type and validity) are manipulated by Fabricate actions. Other records are copied intact. No reports are generated.

3. FINESORT445 — A parameterized (the parameter controls the generation of a key break delimiter) file fabrication. The major task, if no delimiters are required, is to set pocket fields for a mechanical sort procedure. No report is generated.

4. RECON510 — Cards grouped by a key field are validated. Valid groups that will contain three record types are input to Fabricate that combines the three record types into one larger record. Key fields that are identical in all three cards in a group are only copied once.

5. RD022 — A complex program that contains two instances of the Fabricate template (reported in the "Summary Data" as two separate programs i.e., 5.1 and 5.2) one of which makes extensive use of a two-level table to produce summary records. The Fabricate templates are nested within an in-place Update.
C.2.3 Summary Data for Programs Using the Fabricate Template

A. Input/Output Files/Databases/Data Structures

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5.1</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Number of Sources by Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Databases</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tables</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2. Source(s) Access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequential</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Random</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3. Number of Record Types Present in Sources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth of Group Nesting in Sources</td>
<td>N/A</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Source:Source Relationship</td>
<td>N/A</td>
<td>M:1</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1:M</td>
</tr>
<tr>
<td>6. Number of Report</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Report Purposes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Error</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>-Source Detail</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>-Source Summary</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>-Output Detail</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>-Output Summary</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>-Audit Trail</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>8. Number of Outputs by Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Databases</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Number</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5.1</td>
<td>5.2</td>
</tr>
<tr>
<td>----------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
</tr>
</tbody>
</table>

9. Number of Record Types Present in Outputs

<table>
<thead>
<tr>
<th>10. Depth of Group Nesting in Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

11. Output Used for

| -Fabricated | Y | Y | Y | Y | Y | Y |
| -Not Selected | N/A | Y | N/A | N/A | N/A | N/A |

12. Source:Output Relationship

<table>
<thead>
<tr>
<th>Determined by</th>
<th>1:1</th>
<th>1:1</th>
<th>1:1</th>
<th>M:1</th>
<th>M:1</th>
<th>N:M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record type</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Key value</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Position</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Validity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Other Source</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Procedure</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

13. Source Record Selectivity

| All | Y | Y | Y |

B. Fabrication Actions

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5.1</th>
<th>5.2</th>
</tr>
</thead>
</table>

1. For the Selected recs, actions are

<table>
<thead>
<tr>
<th>Ignore</th>
<th>Copy</th>
<th>Report</th>
<th>Manipulate</th>
<th>Data</th>
<th>(see part C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
2. Action Determined By

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5.1</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record Type</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other Sources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Break</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C. Manipulation Actions Present

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5.1</th>
<th>5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Copy of All</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fields of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selected Recs</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Reformatting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-All Fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Selected Fields</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Encode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Combine Several Sources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Maintain Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Loss of Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Combine Records From Single Source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Maintain Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Loss of Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7. Expand
   Single Source

8. Include Program
   Fields  Y  Y

Notes

1. Finesort445 generates a fabricated output for each selected input. In addition, it may (parameterized control) generate group delimiters.

2. RD022-2 the second Fabricate uses a table to store individual inputs and summaries of inputs which generate many outputs.

3. Source records originated in two files and were all placed into a table using complex procedures to determine where the source data was placed. Further complex procedures determined when table values generated output records and which output records were generated.
C.3 Report Generation Template - X

WHEN IS THIS TEMPLATE APPLICABLE?

The Report - X template organizes data residing in file(s)/database(s)/data structure(s) into a human-friendly, human-readable form. The template gathers and processes related records, possibly from multiple sources (both in memory and in secondary storage) with each source possibly containing multiple record types and logical groupings. The processing can be as simple as reformatting input fields or as complex as summarizing a group of data records by manipulating other internal or external structures. The template allows a flexible specification of the report(s) format. This includes a specification of the headings, detail lines and footings, when each of these are to be generated, and how output values are to be formed from related inputs.

Often other complex templates include report semantics. For example, Validate, Update, and Fabricate include extensive report generation capabilities. Less complex templates (e.g. Case or Series) or those dealing with tables do not include report generation semantics within the template definitions. Only programs whose primary purpose is report generation are included in the data summary at the end of the chapter. In the case study sample, Report templates were frequently preceded by either the Fabrication or Selection templates for pulling together the needed fields from
selected records, possibly, from multiple sources. These fabricated records were then sorted to obtain the required ordering of report lines. Of the 6 programs reported here, 4 used Fabricate followed by Sort. One used only Fabricate and another used Select followed by Sort. This Fabricate-Sort-Report pattern is perhaps a candidate for a combined template, except that since they can be linked by a sequential file (the sort file), the required linear sequence is easily constructed.

C.3.1 Template Semantics and Related Case Study Observations

C.3.1.1 Major Components

- **Source(s) of data** to be reported and description of relationships among data items.

- **Report types**.

- **Report contents**, that is, the format of the report, the data to be printed and the means of generating this from the inputs.

The concepts and terminology present in each of the above major components will be defined and discussed next.

C.3.1.2 Source(s) of Data

**Source Data Concepts**

- **Output field completeness** ensuring that all related source data items have been processed before an output field is printed.

- Definition of the **relationship** among records from one source or among records from different sources.
- Selection of source records to be considered for processing.

- Identification of the type of access needed for each source.

Realizations of the Source Data Concepts

The most common technique for realizing output field completeness is to arrange inputs so that all inputs belonging to a group can be accessed and processed before processing begins on another group. This can be accomplished by sorting a single source so as to place related records in adjacent positions, by sorting several sources by compatible sort keys and matching the sources, or by allowing random access to all of the sources, so that related records can be randomly retrieved as needed. Six programs (D200, HOST625, SDL100AL, SDL200AL, SDL300AL, SRS900AL) contained a single external source sorted to place related records in juxtaposition. Three of these contained two internal data sources (single-level tables) that were accessed only on a control break using an indexed linear search (the F template). The tables were used to decode source data for reporting. D200 contained a single-level table that was accessed by a linear search (the G template) invoked only for selected records.

Another technique for ensuring output field completeness is to allow enough output fields (one set for every logical grouping) to be maintained in memory so that, no matter which group a record belongs to, the output fields are available. The algorithm must
determine the identity of the output field. Generation of report lines occurs when there is no further chance of an input record contributing to an output field. This is indicated either by an end-of-file for the input source, or perhaps sooner, by detecting a control break, depending on the source design strategy.

To ensure output field completeness, it is important to understand how the relationships among records is defined. During the case study, the way in which the records in the source were grouped was recorded and produced expected results. In each program a key field(s) was used. Other possibilities included a positional relationship or a delimiter approach.

Not all records present in a source contribute to the printed report. This concept is called "selection of external source." In the case study, four reports used all the records presented to the template. Where the report is preceded by a Fabricate or Select template this is to be expected. In three other reports, records were selected by record type and/or key value.

It is not surprising that database retrieval statements have powerful, flexible selection mechanisms. The segment search arguments of IBM's IMS database system allow for field/key and record type selectivity as do most other DBMS. Other possible selection criteria include record position, validity, and interaction with another source (i.e., the presence of a related record). Sometimes a complex procedure is employed to determine the
selection of a record for reporting.

C.3.1.3 Report Categories

**Concepts**

The following is a list of report categories:

- **Detailed reports** where each selected source input causes the generation of 1 or more output records.

- **Summary reports** where logical groupings of source inputs are used to create summary data about each group, or the group affects report format.

- **Detailed and summary** reports that contain both of the above report types.

- **Concurrently generated** reports where output lines belonging to two or more separate reports are generated during the same pass of the source data.

- **Different report types** that can be generated (not necessarily at the same time) by a single program. A report type is defined as having a unique format, or a unique record selection criteria, or belonging to different report category.

**Related Case Study Observations**

Reports can either consist of detail lines containing fields from individual source records, summary lines containing fields from several source records or most often, a combination of detail and summary lines. Reports that generate titles based on a logical grouping, are called summary even though they may not summarize fields from the record groups. None of the reports in the study were detail-only. However, five reports used logical grouping for
title generation only. Two programs generated summary-only reports where multiple input records were accessed and processed before an output line was generated. Summary output line generation was triggered by the end of a logical record group defined by a change in key value, accessing a delimiter, or satisfying a procedural constraint (e.g., one of the summary fields reaching an established limit).

The Report template needs to identify the condition that causes summary output. Reports use concepts such as identifying the end of logical record group, accessing a delimiter, etc., not only to generate summary lines, but often to generate titles (identifying the new summary criteria). The titles may or may not cause the report to advance to a new page (herein called a page break). Two programs in the study generated titles containing the new grouping criteria. Three other programs tied the control break to a page break, meaning that a page only contained data from one logical group and not several. One program contained a column for identifying the grouping criteria, but only placed a value into this column upon encountering the first detail record of the new group. For subsequent detail records, in the same group, the field was blank.

In order to share commonly needed code or to optimize on file accesses, a program could concurrently generate several separate reports. One program in the study generated concurrent reports with
one report using all of the input records to generate a file-level summarization. The second report selected input records by type and key value. The second report also provided summary lines and page breaks for logical groups defined by key value. It was more common to see report programs that were able to generate reports of several different types, but only one report type per program execution.

Two reports are defined as different if they disagree in any one of the following ways: method of selecting input records, report category (summary, detail, or both), or report format. One program in the study had the ability, under parameterized control, to generate five different report types. The reports differed by the logical grouping criteria to be used in reporting summaries and title generation. They also differed in the mechanism used for the selection of source records.

C.3.1.4 Report Formats

Concepts

Report formatting within the Report template needs to address all of the following semantic concepts:

- report title and report footer presence and contents,
- page header contents and generation criteria,
- columnar heading(s) contents and generation criteria,
- handling of full pages (page breaks),
- handling of changes in logical record groupings (control
breaks) including how to identify the grouping condition,
- use of page footers summarizing data displayed on a page,
- use of control break footers summarizing data belonging to a logical record group,
- report groups which are a collection of output lines that must appear together on an output page,
- use of multiple report columns (each of which can consist of many data columns) per page so that upon completing the reading of a report column the reader must return to the top of the page to continue reading the next report column,
- types of processing used to form output values.

Related Case Study Observations

Reports may contain a single page report title which serves to identify the report, the programming system, the individual requesting the report, and perhaps routing information. None of the programs in the study produced such a lead page. This may be because of system software that controls report printing and automatically adds such pages. Four of the programs generated a final report footing. The report footer assures the reader that he has the entire report and in four programs the report footer amounted to printing "THE END."

Detail report pages usually read from top to bottom across the entire report page. They can thus be generated one print line at a time with no need to return to a line once it is printed. This, in fact, was the practice for all of the reports in the study.

A program in a preliminary study produced a detail and summary
report that was to be written to microfiche. Such reports may break the page (fiche) into report columns. While the concept may be easily explained, the production of such a report is complex because once a line is generated, the algorithm must return to this line for subsequent report columns. Some microfiche generation software assumes this responsibility and prints the report lines in the form needed. If it is done in the application program, a table that holds an entire output page is usually used to hold the report lines until they are printed. When a page break is required, the table is output and reinitialized before processing continues.

A detail page consists of page heading lines, column titles, detail lines, summary lines, page footers and logical group footers, all of which may or may not be used. The study recorded which of these were present and the content of each. It also made note of when various lines were generated.

A major source of variation in report programs is due to the interaction between page breaks and control breaks. If a page is to contain only output from one logical group then a control break causes a page break. If a page may contain several logical groups then there is no connection between page and control breaks. When multiple control groups appear on a single page, they are usually separated by a title line, although one program in the study merely inserts the group identifier into the detail line when a control break occurs.
Page footers (lines summarizing all the data displayed on a page) are usually present when only one control group appears per page. These were not observed in any of the programs in the study.

Control group footers are a popular means for outputting logical group summary data and are, therefore, likely to appear on detail/summary reports. Recall that any report that recognized a logical grouping was called a summary report. Summary reports that use logical grouping only to generate group identification, and do not output summary lines have no need for a control footer.

Report groups defined as a sequence of output lines that must all appear on the same page, can be used to assure that output lines belonging to a single group footer appear together, not partially on one page and continued on the next. To implement a report group, the program must know the size of the report group and then look ahead to ensure that the entire group will fit before a page break. If it is determined that the entire report group will not fit, a page break is forced before generation of the report group begins.

In the study, the type of processing employed to form output values was recorded and included the following:

- The simplest process was a reformatting of all of the fields of the source which was used in three programs. This usually occurred when reporting was preceded by the Fabrication template. Two more programs reformatted only selected fields from input records and either ignored, or used the other fields to determine logical group membership, or to direct additional process actions.

- Because reports were meant to represent information in a
user-friendly manner it was not surprising that decoding of source fields was frequently encountered. This appeared in four of the programs.

- Operations that combined field values were common. The fields to be combined could come from different sources or from many records in a single source or both. All of the programs formed output lines using some type of field combination process. The combination operation could have caused loss of individual field identity such as summing a group of values or could have maintained field identity, usually by copying each of the contributing fields to a separate output field or replacing one field with another.

- It was further possible that a single input could have caused several detail data (not just title or footer lines) report lines to be generated. When a source record contained a repeating group each element of the repeating group could have generated an output line. This was the case with one program.

- Inclusion of program values (i.e., not part of or generated from source data) was the final process category. Examples of such program values would be a line numbering applied to report lines or a row and a column identifier in the multiple report column format.
C.3.2 Overview of Programs Using the Report Template

1. SDL100A1 - a parameterized (to provide the report type and identity of key group to be reported) detail and summary report generator that uses control breaks for title generation only. Additional control structure allows for the reuse of common code among three report types. A six-level key is present in the input file with two record types present. Only one of the two types was selected for reporting.

2. SDL200A1 - summary records are fabricated and output to a sort. The ordered summary records are then reported one input line per each report line. Logical grouping is used for title generation.

3. SDL300A1 - records are selected using a time parameter and released to a sort that forms a single-level key group. The sorted records generate a detailed and a summary report that uses logical grouping for title generation only.

4. HOST625 - uses a single sequential input source containing a single-level delimiter group. A page has a fixed number of lines all of which are summary lines for a single delimiter group. A control break corresponds to a page break.

5. SRS900A1 - a fabricate is used to form records that are sorted and then used to construct a summary report. The fabricate uses two sources, selects records, combines fields from records maintaining integrity of fields and outputs multiple records per input because of a repeating group present in each input. The sort establishes a four-level hierarchy (counting the file as a level) that is input to the summary report. The report accumulates totals on the lowest-level group and does a page break whenever a control break above the lowest-level is encountered.

6. D200 - uses fabricate to selectively tag records and adjust sort keys. Produces two detail-and-summary reports concurrently. The first contains all the original inputs and the second only contains selected records with control group totals. Every input record carries a summary error flag that is decoded into error messages. This can result in 1 source record generating
many output lines. Except for report totals, the reports need not be concurrently generated because all of the first report source records appear first after sorting.
### A. FILES/DATABASE/DATA STRUCTURES

#### 1. Number of Sources by Type

<table>
<thead>
<tr>
<th>Program Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Internal</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 2. External Source Access

<table>
<thead>
<tr>
<th>Record types</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Group</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Levels</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 3. External Source Selectivity:

- ALL: Y Y Y Y Y|N³
- Determined by:
  - Record Type: Y Y
  - Key value: Y N|Y
  - Position: Y
  - Validity: Y
  - Other source: N/A
  - Procedure: N/A

#### 4. Internal Source

<table>
<thead>
<tr>
<th>Number</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Access Template</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>

#### 5. Source Relationship Defined by

- Key group: Y Y Y Y Y Y
- Position: Delimiter

#### 6. Number of Concurrent Reports

<table>
<thead>
<tr>
<th>Report Category</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1³</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Both</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

3³
### 7. Number of Report Types

<table>
<thead>
<tr>
<th>Number of Report Types</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Detail</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1²</td>
<td>1</td>
<td>2³</td>
</tr>
<tr>
<td>-Summary</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-Both</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

### B. REPORT FORMATS

#### 1. Global Concerns

- **Report Heading**
  - N N N N N
- **Report Footing**
  - Y Y Y N Y

#### 2. Page Heading Contents

- **Page NR**
  - Y Y Y Y Y
- **Program ID**
  - Y Y Y Y Y
- **Run Date**
  - Y Y N Y Y
- **Data Date**
  - N N N N N
- **When Generated**
  - Y Y Y Y Y
- **Full page**
  - N N1 N1 Y Y
- **Control break**
  - N N N1 N1 Y Y
- **Group Identity**
  - Y Y Y Y Y
- **.title line**
  - Y Y Y Y Y
- **.every detail line**
  - Y Y Y Y Y
- **.detail line only when changed**
  - Y

#### 3. Footings Used

<table>
<thead>
<tr>
<th>Page</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>N/A</td>
<td>Y</td>
</tr>
</tbody>
</table>
4. Types of Output Actions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatted Copy of</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-All Fields</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>(see A.3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Selected Fields</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>-Encoding</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>-Decoding</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Combine Several Sources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>-Maintain Field</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Loss of Field Integrity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine Records From</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Single Source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Maintain Field</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Integrity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Loss of Field Integrity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand Single Source</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Include Program Fields</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Other than Title</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Use of Report Groups</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>6. Multiple Homogeneous Column Output</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

NOTES:
1. Page headings are not generated but control group identity heading is generated.
2. HOST625 includes a validation template which writes its error report to same file as the summary pattern does, but the error report presents completely the summary so that only summary data reported here.
3. D200 generates two reports but one follows the other. They are not concurrent although the code allows for concurrency. The first report includes all records. the second selects records by type and key value.
C.4 Templates Which Initialize Tables - A, B, J

WHEN ARE THESE TEMPLATES APPLICABLE?

Single-level and multiple-level tables play an important role in the file database processing domain. Tables are used in two ways - as a source of values or as a data structure for synthesizing values. When used as a source of values, once initialized, table values are only retrieved. When used as a data structure for synthesizing values, table values are replaced and/or modified, and then output. Often the table is reused. This means the table is reinitialized in order to begin processing anew. In this section three table initialization templates are explored. A later section will look at table processing templates. Finally, summary data pertaining to both sets of table templates (initialization/reinitialization and processing) will be presented.

Either the A, B, or J templates are appropriate for placing values into a table without regard for any later use of the table. The A and B templates place values sequentially into the table and represent a semantic refinement of the H template which provides for sequential access to a k-level table for an unspecified purpose (that is, H has no specific processing semantics). The A template initializes the table with fixed values (usually zero and/or spaces), while the B template obtains arbitrary values from a sequential file to be used in initializing the table.
The J template places values directly into the table instead of placing them in the next (sequential) location as is the case with the A and B templates. Direct placement requires the data to supply the **identity of the table element** to be affected. The J template is only used when initialization values are being supplied by a file.

Each of the table templates must be used in conjunction with other templates through an embedding process.

Because of the similarity of their function, initializing a table, all three templates are being reported in this section. On the other hand, because of the major differences in their control structures and needed user prompts, the three are defined as separate templates instead of variations of a single template. This is not a hard and fast decision. A single template that allows extensive tuning could be used to consolidate the three templates into one.

The templates may initialize a table by accessing more than one table element at a time. It is possible to initialize a table without traversing the table elements through the use of a single, **table-level move**. Of the tables reported in this section, none were initialized by a single move, but four were **reinitialized** (received values that wrote over existing data) using table-level moves. In terms of the template approach, the table-level move technique is a variation presented to the programmer as an option before choices that utilize table traversals are considered. The template should
contain heuristics that guide the programmer in making the choice. A table level in Cobol is a data item whose description contains an "OCCURS" clause. A table element is any data item that must use a subscript(s) to determine a unique occurrence.

An option for performing initialization, which is not appropriate for reinitialization, is to establish table values at program load time. In Cobol this is accomplished using the "REDEFINES" clause. The redefines option is not available for reinitialization because any process that changes a table value also changes the redefines value because the two areas are one and the same area.

C.4.1 Template Semantics and Related Case Study Observations

The major semantic concepts of table initialization are:

- **Table structure.**
- Structure and content of the source of data to be placed into the table,
- Method of identifying the receiving table element,
- **Table reuse** and possibly, reinitialization.
C.4.1.1 Structure of Cobol Tables

The "OCCURS" clause in Cobol defines a "table level". Like many other languages which employ tables or arrays the structure of elements in a level of a table are identical, but unlike some other popular languages the element can be subdivided into individual data items or additional table-levels. Individual data items can be seen as providing a second dimension to a table-level. Further, if any data item in a table element contains an "OCCURS" clause, a nested table structure results. This nesting can only be three levels deep in current standard Cobol, but several data items within a single table element may contain an "OCCURS" clause. While nesting adds another dimension to a table, definition of several table levels within a single outer table level merely defines independent tables grouped within the outer table. These parallel tables share a common first subscript or index, but are independent beyond that point.

Because of the importance of table structure to initialization, searching, and traversal, the case study recorded the number of nested table levels, and the number of independent table levels present in each table.
C.4.1.2 Structure and Content of the Source of Data to be Placed into the Table

If the data to be placed into the table is arbitrary for each (or many) of the table elements, then the data must be enumerated using a file, or obtained from a data structure present in memory. "REDEFINES" is a viable option if the data is initialized once and subsequently only copied. Other complex and unique processing steps might be used to place data into the table. For example, in the study one program moved elements from one table to another.

Four of the tables in the summary reported used the B template which sequentially places records into the table. One table was not initialized at all. The processing of that table replaced existing data (which was garbage) before any attempt was made to use the table data.

C.4.1.3 Method of Placing Initialization Data into the Table

A fine distinction is made between initialization/reinitialization (which is the focus of the templates under discussion) and processing of the table (described in the next section). Table processing may involve the placement of data into a table as does initialization/reinitialization. The difference hinges on the amount of the table being affected by the operation, the effect on the existing table data, the template that applies and the programmer's intent.
Initialization/reinitialization occurs when:

- values in the entire table are replaced, destroying all the previous contents,

- values in a portion of a table are replaced where the portion is either a recognizable unit (i.e., level) or the table logically becomes that portion of the table with the remainder of the table being ignored, or

- subscripts are reset to an initial value with the intent of reusing the table from that point.

If the scope of change is more restrictive or the existing table values are somehow reflected in the new values (e.g., summation) then this is called table processing. There are situations that still resist this categorization. Take, for example, the resetting of a single data field of each table element to a fixed value and leaving the rest of the table as is. Here, the scope is restricted to a portion of the table — all occurrences of a specific data field (a process criteria), but the values are replaced (a reinitialization criteria). This would be called reinitialization of a portion of the table because the A template is appropriate.

Another interesting situation occurs when a table is used to hold record images. For example, selected records are added to the logical end of the table until some event occurs (e.g., table full). The existing data is replaced, but it is only a limited amount (one element) of the table that is changed. This is a case of table processing.
The technique for placing initialization data into a table depends on when it is to be done, how much is to be initialized at each iteration, the identity (subscript determination) of the element to be initialized, and the source of the initialization data.

Tables are frequently initialized at program load time using the "redefines" mechanism as was the case with 16 tables reported. The other possibility is to wait until execution time which occurred with 11 other tables reported. The design choice between using "REDEFINES" instead of procedural code for initialization is made on criteria such as the data item type, table size and use of the table in other programs. If the data is fixed (e.g., zero and/or spaces), not application-specific, and will not be needed for reinitialization, then "REDEFINES" is useful. In the study twelve tables were initialized with fixed data using "REDEFINES". If a large amount of data is needed, an external medium should be used to contain the data. "REDEFINES" is not appropriate in this case. Since the data to be used by "REDEFINES" must be present at compile time if several programs use the same table, each program will have to have its own copy of the initialization data. This redundancy complicates maintenance and data integrity which usually leads to a decision against using "REDEFINES" for initialization.

If the table may be reinitialized to hold summary data, then a "REDEFINES" is not appropriate. When fixed data is used to
initialize the table and reinitialization is required, then either a single table-level move from one area of memory to the table, or a traversal of the table, moving element-size data from a separate memory location, is appropriate. Table size determines which of the two previous choices will be used. A large table is usually reinitialized by moving element-size data. In this case, reinitialization data for an element needs to be stored in memory and the A template, which places fixed data into the table by looping through an element at a time should be used. In the study four and a half tables reported were initialized using this technique.

One final technique that was observed (but not recommended) relied on the knowledge of how the Cobol MOVE (i.e., left to right character at a time) is implemented. This technique initialized a table by placing the initialization value in the first element and then moving the table to itself. This technique is reported as a "Ripple Move".

In the previous paragraphs the placing of data was discussed without regard to the identity of the target element. In another variation, the initialization data carries the identity of the element where it should be placed. Such a scheme allows table elements to be processed directly (without a search) based on the element identity. The J template allows for this form of initialization.
C.4.1.4 Table Reuse and Reinitialization

In the study fifteen tables were reused, that is the existing values were discarded and replaced with new data before processing continued. On the other hand eighteen tables were not reused and, therefore, did not receive new values once initial values were established. These eighteen tables were either used solely as a source of values or the processing of the table required the results be available for the entire execution of the program (e.g., file-level summary fields).

In three cases where reinitialization was used, the tables used fixed values and table traversal — the A template.

Four tables received application specific data through a table-level move. This technique is used when there is a need to access individual characters which comprise a larger string.

Seven programs reused the tables merely by resetting the subscript that pointed to the end of the table, back to 1.

Table reuse was most frequently triggered by a new record. This was the case with seven tables. Six tables were reused when the program encountered the end of a logical group of records. One tables were used to hold data from different files and were reused upon detecting an end-of-file condition. One table used a combination of record type or end-of-file to trigger reuse.
C.5 Templates That Process Tables – H, F, G, P

WHEN ARE THESE TEMPLATES APPLICABLE?

The common patterns of access to table elements are covered by the above four templates. All of the tables reported used one or more of the above access templates to process table data. There were programs in the study that defined tables, but did not use the tables.

C.5.1 Template Semantics and Related Case Study Observations

The H template provides sequential access to every element of a table or to every element of the table at selected levels. The primary use of this is to output a table to a file either to save the table for later processing or to sort the elements of the table.

The F, G and P templates contain search semantics. The first two are linear searches (one requires the table to be indexed and the other does not), and, for completeness, P implements the binary search.

The data collection during the study was based on how data was placed into the table and retrieved from the table.
C.5.1.1 Placing Data into the Table

When placing data into the table, the **effect on the existing table values** as well as the **means of identifying the table element** to be changed are important and are discussed next.

The table is often used to store record images as part of a partitioning process. This processing occurred in nine tables in the study. A table-level move is also used for the purpose of gaining character-level access, and is reported as a reinitialization action. Ten tables contained no data other than that present after initialization.

Existing table values can be **augmented** by a new value. Only one table reported was used in this way to hold accumulated values.

If the element identity is **known** to the program through a fixed procedure or is available in another data item, then it is possible to directly access the element. Direct access was used for four of the tables reported. The element identity can be the **next element**. Here the identity is maintained in a counter variable. Next element access was the case in seven of the tables reported. Two tables used a **linear search** to determine where to place values.
C.5.1.2 Obtaining Data from the Table

Table access to obtain values from the table can involve the entire table as was the case in six tables reported or it can be on an element by element basis as occurred in twenty-five tables reported.

The H pattern can access the entire table one element at a time and was used in four tables in the study. Direct access implies knowledge of the element identity (by subscript) so that that element is immediately accessed. This occurred in five and a half tables.

Searching can also be used to identify the program element. Seven tables relied on an indexed linear search (Cobol SEARCH verb) while eight and a half used a subscript-oriented linear search.

A final issue with linear searching is the completion criteria for the search. It covers the question: "Should the search stop upon finding the first match or should all elements successfully meeting the search criteria be identified?"
C.5.1.3 Summary Overview of Programs That Use Table Templates

Unique programs from the sample that used table template(s) are listed with a brief comment.

1. D100 — This is an extensive validation program that uses tables both as a source of test objects during validate and as a source of output values.

2. HOST645 — This is a complex combination of Validate, and Fabricate that uses tables for image storage, character access, decoding and summary storage. A unique aspect is a match of two tables where the first table is searched and, upon finding a qualifying element, the second table is searched for a matching entry.

3. SDL010A1 — A validate program that uses three indexed tables for test objects in set membership tests.

4. D160 — An update program that collects logical groups of transactions into a table and generates (adds) records to the master only if the master has a record that matches the transaction group. The table is not needed because a match between the transaction and master could be used.

5. RD610 — Two types of tables are present; one type keeps track of surviving banks and duplicate account numbers, the other type merely provides variable length records. The first type is searched and data is moved between tables.

6. VGA960 — The program contains a table-level that contains two other levels both of which are nested within the first, i.e., two tables nested one level deep not nested within each other. A routine is included to load the table from a database using sequential placement at the current table level (i.e., the B template). At the same time two fields of the table are initialized with fixed values as provided by the A template. After initialization, values are only taken out of the table by first searching the outer level and then directly accessing the first inner table using a parallel location for the first subscript and a known value for the second subscript.
### C.5.2 Summary Data for Programs Using Table Templates

<table>
<thead>
<tr>
<th></th>
<th>Program Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1   2   3   4   5   6</td>
</tr>
</tbody>
</table>

1. **NR Tables Used**
   - 1
   - 2
   - 3
   - >3

2. **Nested Levels**
   - 1
   - 2
   - 3
   - >3

3. **INIT by**
   - **Fixed Values Using**
     - Redefines 3
     - Table Level Move
     - A Template Ripple Move
       - 1
       - 1

   - **Unique Data Using**
     - Redefines 2
     - Table Level Move
     - J Template B Template
       - 1
       - 1

   - **Not Initialized**
     - 1

4. **Reused**
   - Yes 0 8 0 1 6 0
   - No  5 7 3 0 2 1

5. **Reused How**
   - N/A

   - **Reinit Fixed Data Using**
     - Table level Move
     - A Template Ripple Move
       - 3
       - 1
<table>
<thead>
<tr>
<th>Program Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>Reinit Unique Data Using Table Level</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Reset Subscript</td>
</tr>
<tr>
<td>6. Reuse Trigger N/A</td>
</tr>
<tr>
<td>Control Break</td>
</tr>
<tr>
<td>EOF</td>
</tr>
<tr>
<td>New Rec</td>
</tr>
<tr>
<td>New Rec or EOF</td>
</tr>
<tr>
<td>7. Use Placing Data Into Table</td>
</tr>
<tr>
<td>-N/A</td>
</tr>
<tr>
<td>-Insert Unique Data Into Element</td>
</tr>
<tr>
<td>-Augment Current Element Data</td>
</tr>
<tr>
<td>-Element Identity</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>8. Use Accessing Values From</td>
</tr>
<tr>
<td>-Entire Table</td>
</tr>
<tr>
<td>-Element</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
9. Purpose for Table

-Decode  1  2
-Set membership  4  3  2
-Char Access  4
-Save Images  7
-Accumulate Values  1
-Summary Flag  2
-Supply Update Values  1

NOTES

1. Part of the table was initialized with fixed values using the A template, and part with unique values using the B template.

2. Table is partially initialized from a file and partially by fixed null. This is really two, separate parallel tables. The first part is searched which yields the subscript to access the second table first location. Therefore, 1/2 G, 1/2 direct.
C.6 The Update Template - U

WHEN IS THIS TEMPLATE APPLICABLE?

Files/databases contain data that model real world objects or events. The real world is changing and, to be useful, a file/database must reflect those changes. The purpose of the Update template is to modify the data in a file/database (called master) in order to keep it current. The changes to be made are called transactions and can be applied to the master file as they occur (on-line) or collected in a batch to be applied at a later time. Batched transactions can be applied on a periodic basis (daily, weekly, etc.), when a sufficiently large quantity is obtained to make sequential I/O economic, or whenever the data in the master file is about to be used and needs to be current.

C.6.1 Template Semantics and Related Case Study Observations

C.6.1.1 Major Components

As discussed above, the major components of the template to be discussed in the rest of this section are:

- Sources and destinations for the data.
- Transaction type data.
- Master type data.
- Log files.
- Reporting.
The following categories of data can be processed by the Update template.

- Transaction data that directs the updating of the master file.
- Master data about a collection of entities.
- Log data captures transaction and/or master record images providing an audit trail or a means of backing out (undoing the update).
- Report data detailing update errors but also serving to identify what was done during the update in a manner similar to the log file, but in report form.

The concepts for handling the above collections of data will be discussed next. This discussion will serve to define terminology and concepts present in the Update template for implementing update programs.

C.6.1.2 Transaction Data

THE TRANSACTION CONCEPT

Transaction data:

- identifies (using a master-identity) the master records that must be modified,
- identifies the processing action(s) to be applied to the master record, and (possibly)
- provides values to be used by the processing actions.
TRANSACTION CONCEPT REALIZATIONS

Transaction data can be provided either

- with files (transaction-in and/or transaction-out) if
  enumeration of transactions is required, or

- without files, by having fixed possibly parameterized
  processing actions and fixed or procedurally determined
  master identities.

1. Transaction-in

The transaction-in file is the source of transactions to
be applied to the master file. The following terms
characterize the ways in which the transaction-in data is
used in the Update template.

1.1 GROUPINGS OF DATA IN TRANSACTION-IN

The update can be preceded by a sort of the transaction-
in file. Sorting accomplishes several objectives. Where
no more than 1 transaction record exists for each master
record and a transaction record can be applied to no more
than 1 master record (hereafter referred to as 1:1
relationship), the sorting of the transaction file allows
a matching algorithm to be used. Where the master-
transaction relationship is 1:N (i.e., many transactions
could apply to a single master), sorting of the
transaction-in file collects the transactions into groups
by key. A final reason for sorting is to group
transactions (possibly within a key group) by action
type.

All of the update programs reported used a single
transaction-in file that was accessed sequentially. In
five of these programs (D120, D180, SDL020A1, RECON550
and RD685) the transaction file enumerated both the
master identities and corresponding actions. In VGA960
only the master identities were enumerated. The action
was fixed. IMS1 used the transaction file to pass a
single fixed master identity parameter.

In three programs, D120, D180, and RECON550, action type
was used as a sort key. Deletes preceded Adds which
preceded Changes in all three programs. In the other programs the submittal time was used to order transactions within key groups either overtly through a time stamp or sequence field, or by assuming a stable sort (given equal sort keys first read is first output). Transactions ordered by action type instead of submittal time have a profound effect on allowable sequences of transactions pertaining to a single master record. D180, which allowed any legal combination of actions to be submitted, collected deletes before adds before changes. Consequently, it was impossible to delete a record that was added in the same transaction file. Attempting to do so could have resulted in an error "attempting to delete a nonexistent record."

1.2 SELECTION OF DATA FOR TRANSACTION-IN

All the transactions present in the transaction-in file do not have to be considered for the updating process. Typically, selection is necessary when both valid and invalid transactions are in a single file, or when a single file contains transactions for updating several different master files.

In three of the programs in the study, transactions were screened or selected before being used. If they did not meet the selection criteria, another transaction was read before any updating was considered. Selection was based on testing a validity flag field in the transaction (set during an earlier validate program) or on the record type of the transaction. One program only selected the first 7 transaction records because they were placed into a table and used repeatedly.

1.3 INDICATING MASTER-IDENTITY AND ACTION

At times either master-identity or action is fixed or determined procedurally and the transaction-in file which only enumerates variable (not fixed) data is not used.

Fixed-action, enumerated-master-identity updates such as "Purge" programs (fixed action delete) and "Mark" programs (fixed action change) commonly occur.

Two programs RD685 and VGA960 fixed the action and
enumerated master identity resulting in a transaction-in file containing master key fields only.

If the transaction-in file only enumerates one type of data, it is usually the master identity not the action. This is because of the possible size of the target set. The number of master record keys to be identified could be quite large while the number of actions is only three (delete, add, change) which, if we allow all possible combinations (of 1, 2, or 3 actions), is only 10. An action enumerated transaction-in file would serve to parameterize the possible functions in the program.

If master identity and action are both fixed or determined procedurally then enumeration is unnecessary and the transaction-in file is not needed. A purge program (fixed action delete) that tests a delete flag in each master record (procedurally determining master identity) would be one example of an update without a transaction-in file. It should be noted, however, that the requirement for master identity and action are not absent, but merely being served without a file. "Mark" programs, which set delete flags or time stamp master records, are fixed-action (change) fixed-master-identity updates.

IMS1 is a purge type update (fixed-action delete) using procedurally determined master identity by testing the expiration date field of low-level segments and deleting out-of-date segments. Further, root segments with no subordinate segments are also deleted.

2. Transaction-out

While the use of the transaction-in file as a source of transactions is common, the transaction-out file which is designed for disposition of transactions is less prevalent. The transaction-out file(s) contains transaction records and is created during the update process for one or more of the following reasons: to provide a backup copy of the input transactions selected for update, to move the transaction file to a faster medium (e.g., card to disk, or tape to disk), for reflecting manipulation of transaction record fields such as error flags, or for partitioning of transactions by validity, record type or key group.

Program D180 from the study wrote all input transactions
to the transaction-out file so that the output transactions would contain updated error flags reflecting any errors discovered during update. In the original RD685 program, a backup copy of the transaction-in file was made using the transaction-out facility. RD685 contained a "Move" action that caused a delete from one master file and an add of the deleted record to a second master file. The proposed template solution to this unique application was to link two update patterns together using the transaction-out file. The first update would delete appropriate master records from the first master file creating an add transaction from the deleted record and writing it to the transaction-out file. The transaction-out file then became the transaction-in file for the second update which was a fixed-action (add) enumerated-master-identity update.

3. Master-Identity

Master-identity is the term used for all the mechanisms that identify a master record to be effected by update. The master-identity mechanisms are based on either 1) enumerated key fields or 2) a fixed criteria. A transaction-in file containing master key fields can either be processed using a sequential match algorithm or can supply keys to be used in randomly accessing a master file. The key in either case can be a single field testable by a single expression or a group of fields (multi-field key).

Four programs used a multi-field key. Two programs processed the multi-field key by testing multiple conditions. The other two constructed a composite key by concatenating key fields left to right in order of significance. This key composition step simplifies the control structure.

Fixed-master-identity criteria can be as simple as including all master records, can test a single master field (e.g., date, or delete flag) or could involve a complex set of procedures such as determining if a segment is root only. Only IMS1 used fixed-master-identity, but it involved two of the above techniques (i.e., expiration date field and root only segment).

4. Action

Three types of actions are used in update programs:
a. delete - removing or marking a master record as not being part of the master file.

b. add - creating a new master record to be included in the master file.

c. change - modifying 1 or more non-key fields from an existing master record.

When the action is enumerated, the action type is always identified by a coded field within the transaction. Another possibility is to provide a record group delimiter (all adds together), but this would make a sequence of transaction for one master difficult to implement. The delimiter technique was not observed in the programs.

These three actions might all be allowed in a single update program as was the case with (D120, D180, and SDL020A1 programs from the study) or might appear in some combination excluding 1 or 2 actions. For example, VGA960 and IMSI were delete only update programs, and neither RECON550 nor RD685 allow a separate change action. A related issue that will be explored later is the allowable, effective transaction sequences that could be specified for a single master record.

The delete action is generally realized in one of three ways. Where a new master is being created, writing of the deleted master record could be inhibited or it could be written with a delete indicator set. If the system access method (e.g., VSAM) permits, and an in-place update is being executed, the deleted record can be physically removed from the updated master. Five programs from the study chose to inhibit writing of a deleted master, one chose to set a delete flag, but still wrote the deleted master, and one chose to physically remove the deleted record.

The changing of a field in an existing master record could be done by destroying the existing field (replacement) or by augmenting the existing master field value by combining it in some way (e.g., addition or subtraction) with the change field value. Replacement of an entire master record is implemented using a two-transaction sequence - deletion of the existing record followed by addition of a new record containing a copy of all the existing master record values except those being
replaced. Augmentation type changes can also be implemented with a delete-add sequence. The user again must know all of the existing master values, perform the augmentation operation and construct the add transaction. As part of the application requirements RECON550 did expect users to want to change master records, but since no separate change action was defined the user was directed (forced) to submit a delete-add sequence. Because of the need to know existing master field values, having to copy these on an add transaction and submitting 2 transactions, this method of implementing a change is seen as inconvenient for the user and error prone. It does, however, simplify program logic.

Since a change transaction seeks to selectively modify individual fields of a record, the problem of identifying which fields are to be changed is encountered. Two design solutions are used. One, called "keyword," requires the field to be named or its ordinal position (i.e., fifth field) to be given in addition to stating the new value. Using the keyword technique, a transaction can change any number of fields and the fields can be specified in any order. The other technique called positional or image requires the change transaction to list new values in a fixed sequence and to provide a mechanism for indicating that a field is not to be changed. Job control languages (JCL) uses a comma delimiter to separate positional fields, but this requires a scanning of the input and a counting of commas to determine field number. Most often a fixed record format, allowing space for each master field, is used. This method requires the use of a "null" (usually spaces) value to indicate no change to a field.

5. Sequences of Actions Dealing with a Single Master Record

It is possible that a transaction-in file might have more than one action that pertains to a single master key. Most of the programs studied recognized the following errors which may be caused by a sequence of transactions for 1 master record:

<table>
<thead>
<tr>
<th>Error:</th>
<th>Multiple Action Sequence Causing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deletion of nonexisting master record</td>
<td>delete followed by a delete with no intervening add</td>
</tr>
<tr>
<td>Addition of an already</td>
<td>add followed by an add with</td>
</tr>
</tbody>
</table>
existing master record no intervening delete
Change of a non-existing record change following a delete with no intervening add

The state-transition matrix in Table 20 defines all of the allowable single-master record transaction sequences. The initial state depends on the presence of a master record in the master file.

**Table 20: State-Transaction Matrix Defining Legal Transaction Sequences for a Single Master Key**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Change</td>
</tr>
<tr>
<td>1. Master Defined</td>
<td>3</td>
</tr>
<tr>
<td>2. Master Not Defined</td>
<td>1</td>
</tr>
<tr>
<td>3. Error</td>
<td>3</td>
</tr>
</tbody>
</table>

The numbers contained in the matrix identify the next state.

6. Relationships Between Transaction and Master Records

This is a discussion of the number of transactions that are related to a single master or the number of master records that are related to a single transaction. Later the grouping criteria used when a group of either master or transactions are present will be discussed.

By far, the most popular form of relationship observed was 1 master related to N transactions which was observed
in four programs D120, D180, RECON550, and SDL020A1. Although the transaction files of both D120 and SDL020A1 were sorted and did organize the transactions by key group, no local loops were included in the update code to take advantage of this grouping. SDL020A1, which was an in-place update using random access on the master file, repeatedly wrote and accessed the master file while processing a transaction group. D120 used a global retest of the transaction and master keys instead of a local loop to extract the transaction key group.

When the transaction is fixed, the relationship between master and transaction becomes (M:1 this occurred in IMS1). The M:1 relationship also occurred when the action was fixed but the master-identity enumerated. Both RD685 and V6A960 used the transaction file to hold group identifiers for collections of master records. VGA960 did not employ a local loop to process all master records in a group, while RD685 used a local loop to process all master records within a key group. No other relationships such as 1:1, containing a transaction for each master record, were observed.

C.6.1.3 Master Data.

THE MASTER CONCEPT.

Master data must be identifiable, using the previously defined "master-identity" concept, if the updating is to affect the appropriate master record.

REALIZATIONS OF THE MASTER CONCEPT

The master file(s) can be updated in-place (if a single file serves as the source of records and also the updated master file), or updated by producing a separate file to contain the updated master. In all but 3 programs a single, sequentially accessed file served as the source of master records and a separate output file
served as the updated master. RD685 used 7 parallel sequential files to contain master data and 7 separate parallel output files as the updated master. SDL020A1 and IMS1 both did an in-place update, i.e., only 1 master file served as input and output. SDL020A1 accesses records randomly while IMS1 accessed records sequentially as well as randomly.

Two important aspects of the master source file which were reflected in several programs were multiple record types and selection of master records to be considered for updating. The existence of multiple record types complicates the control logic because, frequently, each record type requires a different add and change procedure. Delete is usually uniform across record types. Frequently, not all of the master records present in the master source are available for updating. In D120 master records carried a logical delete flag which, when set, caused the next master to be read. IMS1, through the segment search arguments (SSA), selectively retrieves records (segments) by segment type.

C.6.1.4 Log File

THE LOG FILE CONCEPT

Log or audit files which can be used to restore or recreate the master following execution of the update, became popular with database system software. In fact, in most DBMS systems, the decision to use a log file is a system-wide decision and not based
on the individual applications. Log files store "before" images (i.e., master record contents before any changes) and possibly "after" images (i.e., updated master records).

The use of a separate file for the updated master leaves the master source unaltered, thus eliminating the need for reconstructing the master source. In-place-delete-only updates can be undone if deleted records are only flagged and not physically removed.

C.6.1.5 Reporting

THE REPORTING CONCEPT

Reports serve several purposes. In order of frequency of occurrence the purposes observed are: 1) to communicate errors, 2) to list each old master record that is changed, possibly including summary lines, 3) to list each transaction record possibly including summary lines, 4) to produce check sums by action type, and 5) to list all old master records (not only those updated). All of the programs except one (D180) produced a report while updating the master. VGA960 and IMS1, both fixed-action-delete programs, did not check for transaction errors and, therefore, generated no error report. This is not surprising since the only applicable error - attempting to delete a nonexistent record - was not needed when the master identity was either based on a group key (VGA960) or on fields within the master record itself (IMS1).
C.6.2 Summary Overview of Programs Using the Update Template

1. **D120** — A classic update of a sequential master file with the following exceptions so: 1) transactions are sorted within a key group such that Deletes are first followed by Adds, then followed by Changes. This order makes the deletion of an Added record impossible within one program run. In fact, it may cause an error (deletion of nonexistent record) to be generated, 2) only 1 field can be changed, and 3) master records can be physically present but not available for update.

2. **D180** — A later update program from the same system as D120. This, again, is the classic sequential file update with the following interesting points: 1) master records may be physically present yet not available for update, 2) transaction records that carry a validity flag can likewise be present in the transaction file but because they are invalid (as indicated by the flag) are skipped, and 3) there are 24 different types of master and transaction records which complicates the update logic by requiring numerous case statements to determine record type.

3. **RD685** — Use of the update template is complicated by two unique aspects: 1) the master data is represented as a set of 7 parallel sequential files, 2) selected master records are always deleted from 1 group of 7 master files and, depending on the transaction, they may be added to a new or existing group of 7 master files, 3) because of the need to process the transaction file 7 times, it is placed into a table to save I/O's, and 4) the update allows only a single action type per program execution. When the transaction calls for a "move" it in effect means a delete from one master file group and an add to another master file group. Both actions "Delete" (which is the classic delete action) and "Move" (which is combination of classic delete and add actions) are implemented with exactly the same code, i.e., both are treated as delete and add. If the "Delete" is actually required, the newly created file is destroyed. This is the reason for limiting each execution to a single action type.

4. **RECON550** — A sequential file update that does not provide a separate change action. The change must be brought about by the user submitting a delete followed by
an add. In order to preserve tape drives, the updated master is written to disk and later (when a tape drive is available) copied to tape.

5. SDL020A1 -- An in-place update that ignores key grouping of transaction records and, therefore, reads and rewrites the master record (possibly with deletion flag) for each transaction. The master file contains a date record that identifies the last time the file was updated. Transactions that have a date earlier than the master update date are not selected for processing.

6. IMS1 -- A fixed-action, (parameterized) fixed-master-identity in-place update of an IMS hierarchy containing 4 segment types. Root only hierarchies those where all subordinate segments have been deleted are also deleted. Random access is necessary to accomplish the "root only" delete because the root segment has been passed by, in the sequential processing, before the decision to delete can be made. A random access is used to back up to the root.

7. VGA960 -- A fixed-action (delete) enumerated-master-identity update of a sequential master file. The old master is not changed but a new master is created along with a report of deleted records.

The quantitative results of the template-centered study are summarized in the next section and are presented using the terms discussed in the previous section.
### C.6.3 Summary Data for Programs Using the Update Template

#### A. Files/Databases/Data Structures

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1 2 3 4 5 6 7</th>
</tr>
</thead>
</table>

1. Transaction Source

<table>
<thead>
<tr>
<th>Number</th>
<th>1 2 3 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR Rec types</td>
<td>5 24 1 5 2 1 1</td>
</tr>
<tr>
<td>NR Key Groups</td>
<td>1 1 0 0 0 1 0</td>
</tr>
<tr>
<td>Sorted in Pgm.</td>
<td>Y Y Y Y N N N</td>
</tr>
<tr>
<td>Avail. Trans.</td>
<td>Valid First All Select All Y</td>
</tr>
<tr>
<td>Master</td>
<td>Y Y Y Y Y Y N</td>
</tr>
<tr>
<td>Identity</td>
<td>Enumerate</td>
</tr>
<tr>
<td>Action</td>
<td>Y Y Y Y Y N N</td>
</tr>
<tr>
<td>Enumerate</td>
<td></td>
</tr>
</tbody>
</table>

2. Transaction Output

<table>
<thead>
<tr>
<th>Number</th>
<th>0 1 1 0 0 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>N N N N</td>
</tr>
<tr>
<td>Modification</td>
<td>Error flag</td>
</tr>
<tr>
<td>Fabricated</td>
<td></td>
</tr>
</tbody>
</table>

3. Master Source

<table>
<thead>
<tr>
<th>Number</th>
<th>1 1 7 1 1 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>NR Rec Types</td>
<td>2 7 7 1 1 1 1</td>
</tr>
<tr>
<td>NR Key Group</td>
<td>0 1 0 2 0 1 3</td>
</tr>
<tr>
<td>Levels Available</td>
<td>Not All All All All All</td>
</tr>
<tr>
<td>Master recs</td>
<td>deleted Selected by SSA</td>
</tr>
</tbody>
</table>

4. Master Out

<table>
<thead>
<tr>
<th>In-place update</th>
<th>N N N N Y Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Seq. Seq. Seq. N/A Seq. N/A</td>
</tr>
<tr>
<td>Rec. Types</td>
<td>2 7 14 1 1 1</td>
</tr>
<tr>
<td>Groups</td>
<td>0 1 0 2 1</td>
</tr>
</tbody>
</table>
### 5. LOG Files

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Master Before</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Master After</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Trans.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 6. Reports

<table>
<thead>
<tr>
<th>Number</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- error</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Transaction</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Transaction</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- All Master</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- All Master</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Changed Master</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Changed Master</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Action Details</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B. Transaction Details

#### 1. Actions

<table>
<thead>
<tr>
<th>Number</th>
<th>3</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete - D</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Add - A</td>
<td>Y</td>
<td>Y</td>
<td>Note 2</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Change - C</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>Note 3</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Combination</td>
<td>N</td>
<td>N</td>
<td>D,A</td>
<td>D,A</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

#### 2. Action Type Indicator

<table>
<thead>
<tr>
<th>Field in Record</th>
<th>Y</th>
<th>Y</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group Delimiter</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed Assumed</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameterized</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
### 3. Master-Transaction Relation

<table>
<thead>
<tr>
<th>Program Numbers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

- grouped by
  - key
    - delimiter

- local loops
  - N

### 4. Comparison Key

<table>
<thead>
<tr>
<th>None</th>
<th>Single Field</th>
<th>Multi-field</th>
<th>Multi-field condensed key</th>
<th>Individual key</th>
<th>Field test</th>
<th>Enumerated</th>
<th>Procedurally determined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

### 5. Delete Action

<table>
<thead>
<tr>
<th>Write with flag</th>
<th>No write</th>
<th>Physically Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Program Numbers

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

6. Change Action

Keyword replacement

augmentation Positional Y Y Y
replacement 1 34 11
augmentation 0 18 0

7. Add Action no data collected other than that reported in B.1

8. Error Testing None None

Delete non-existing Y Y Y Y Y
Add Existing Y Y N Y Y
Change non-existing Y Y N/A N/A Y

Integrated Validate Tasks Y N Y Y
Out-of-date transaction None N/A N/A N/A Y

9. Ordering of Transactions N/A

By Master Keys 2 1 1 1 1 1
By Order Submitted 2 2 2 2
By Action Type 3 2 2
By Record Type 1

10. Transaction Sequences

Any Legal
Restrictions None
- No delete Y Y Y
- Delete-Add Y
- No Add following a Delete Y
- Delete Only Y Y
- A deleted rec may be changed Y

Notes:

1. Transaction file placed into a table was sequentially reused 7 times, once for each of the parallel master files.

2. The add does not exist as a stand alone action but it is used in combination with a delete from one group and an add to a second
group. It is, therefore, listed as a combined action; call it move.

3. No individual change action is allowed for, but the program allows a delete followed by an add which amounts to a change (positional replacement) of all of the fields.

4. Keys of deleted segments are output to a file which is used as a transaction file to update other masters.
Appendix D.

Template Definitions

D.1 Template Definition for: (CASE - C)

PURPOSE

(Case) provides 1 or more mutually exclusive process branches.

COMPOSITION

(Case) is a primitive, that is, it does not include any other templates in its definition.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Ordered Sequence of Test Conditions - Both the ordering and the text of the conditions are essential.

2. Process Actions - Each test condition has a corresponding process action branch where programmer written process code will be located. This code will be executed when the corresponding test is true and no preceding test in the case is true. The final test will have both a true process action branch, and a false process action branch. The false process action branch is only executed when all of the tests in the case fail.
D.2 Template Definition for: (Series - L)

PURPOSE

The (Series) template provides a series of one or more process branches, each of which resides in the true branch of an IF statement. Each succeeding condition is evaluated regardless of the outcome of previous conditions.

COMPOSITION

(Series) is a primitive, that is, it does not include any other templates in its definition.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Ordered Sequence of Test Conditions - the ordering is necessary because, although the evaluations of conditions are independent of one another, the true or false value may be affected by earlier tests or the process may depend on earlier tests.

2. Process Actions - each condition governs the execution of a process branch. The branch is executed when the condition is true.
D.3 Template Definition for: (Select - S)

PURPOSE

(Select) is a commonly occurring special instance of (CASE) nested within a single loop. It repeatedly executes a case structure until the loop escape condition is met.

COMPOSITION - C,I

Select utilizes the full semantics of (CASE) but restricts (Generalized Traversal) to a single loop. Although unlikely, it is possible that the source access semantics of (Generalized Traversal) will not be used in (Select).

SEMANTICS AND CLASSIFICATION

Essential Independent

1. Ordered sequence of test conditions - see (CASE)

2. Process actions - see (CASE)

3. A single loop traversal - see (Generalized Traversal) with the semantic restriction of only allowing a single loop which may or may not traverse a data structure.
D.4 Template Definition for: (Series with Iterator - E)

PURPOSE

(Series with Iterator) is a commonly occurring special case of (Series) nested within a single loop. Although it is conceivable that the loop would not be traversing either a table or external structure in which case the source access semantics of the I template would not be needed, it is unlikely.

COMPOSITION - L,I

As the name implies, (Series with Iterator) is a combination of the (Series) template with the (Generalized Traversal - I) template. It is a special case because it limits the semantics of the I template to a single loop.

SEMANTICS AND CLASSIFICATION

Essential Independent

1. Ordered sequence of test conditions - see (Series)
2. Process Actions - see (Series)
3. A single loop traversal - see (Generalized Traversal) with the semantic restriction of only allowing a single loop that can traverse a file, table, database or no data structure at all.
D.5 Template Definition for: (Generalized Traversal - I)

PURPOSE

This template allows for the description of the relationship(s) among any number of data sources, both in memory and/or secondary storage, when each source should be accessed and the form of the access statement needed. It is the source of data structure traversal semantics for all of the other templates.

COMPOSITION - C

The structures needed are sequence, iteration, choice, refinement, and selection. The first four are defined in the template itself. Selection is a use of the (Case) template.

SEMANTICS AND CLASSIFICATION

Since multiple sources are common, the source type and access statements needed are also defined. The semantics permit the description of a Jackson program structure diagram.

Essential Dependent

1. **Source(s) Identification** - the name of each data structure to be traversed.

2. **Source(s) Type** - for each source named above, the type of the source, i.e., sequential file, indexed sequential file, relative file, table, database and DBMS software.

3. **Relationship(s)** - among the sources described in the Jackson methodology.
D.6 Template Definition for: (Sort - 0)

PURPOSE

The (Sort) template collects syntactic data to be used in generating a Cobol Sort verb which uses a source file and produces sorted records in an output file. For those familiar with the sort verb in Cobol, the Input and Output procedures are not utilized because their functions can be served by a linear sequence of templates most commonly (Fabricate) or (Report).

COMPOSITION

No other templates are used in the definition of Sort, assuming the Cobol SORT verb is used. If a different sort algorithm is used requiring source language code to be generated to implement the sort, several other templates could become involved.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Identification of the source and destination files which may be one in the same and which both have identical record formats and numbers of records.

2. Definition of key fields (location and size) to be used in sorting and the order of importance of the keys.

3. Definition of the order (ascending or descending) to be used for each key field.
D.7 Template Definition for: (Update – U)

PURPOSE

(Update) applies transactions to an existing information model residing in a master in order to bring the master up to date.

COMPOSITION – I,X

(Update) relies on (Generalized Traversal) to traverse the master and transaction sources and to bring related master and transaction records together; (Report) supplies semantics for generating all of the possible reports.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Master data source(s) – the semantics of the (Generalized Traversal) (transformed to identify the file purpose) is used to generate a traversal of the master data sources, as well as the transaction data source(s).

2. Transactions – that may or may not reside in a file define the processing actions and the identity of master records to be processed. When transaction data is enumerated in a data structure, (Generalized Traversal) semantics are used to generate the needed source code. Both the master identity and the processing action(s) can be fixed, in which case a transaction source and traversal are not needed.

3. Processing Actions are delete, add and change and can be used in any combination within the program, but there are restrictions on allowable sequences of actions for a single master record.

4. Error conditions – three error conditions are tested for, depending on the use of corresponding process actions.
They are:

- deletion of nonexisting master,
- addition of existing master,
- change of nonexisting master.

Inessential Dependent

1. **Log files** – these are output files that allow for recovery of the master file prior to the update. This can be accomplished by storing "before" and possibly "after" images of the master record and related transactions.

2. **Reporting of update activities** – reports can be generated for any one or a combination of the following purposes:

   - to identify update errors,
   - to list old master records possibly with summary data,
   - to list transaction records with or without corresponding master records and summaries,
   - to list summaries by update actions,
   - to list all old master records not only those updated.

3. **Transaction outfile** – either contains images of the transactions used in updating the master or possibly transaction records created during this update to become input transactions for a second update. It provides a means of keeping multiple file masters updated.
D.8 Template Definition for: (REPORT - X)

PURPOSE

The (Report) template traverses one or more sources of data, provides process code locations so that output data can be formed from selected input records in accordance with relationships among the sources. Further (Report) includes semantics that deal with the format of the output report.

COMPOSITION - I, H, F, G, P

The (Generalized Traversal) template provides the semantics needed to traverse one or more sources which are either stored in memory (tables) or in secondary storage (files/databases). This traversal allows for selection of source records as well as process code locations where related data fields can be manipulated to form output fields.

The (Table - H, F, G, P) templates provide local access to tables used as sources of data to be reported.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Report Purpose

2. Source(s) of Data - records whose data will appear on the report.

3. Output field completeness - individual output fields are not released to the report(s) until all of the sources(s)
and all of the records from those source(s) related to
the output fields have been processed.

4. Report Categories Essential Dependent Choice (one of
these three is in effect)

- Detailed report - one or more output lines generated
for every input record selected for processing.

- Summary report - a logical grouping, hierarchy of
logical groupings or sequence of logical groupings
is used as a basis for generating output records.

- Detailed and Summary - a single report that contains
both of the above concurrently generated for a
single report.

5. Report Format - is concerned with the actual layouts of
the report.

- Page Format including:

  * page header contents includes - report date,
data date, program and/or system
  identification, page numbering, title contents
  and generation criteria including page breaks
  and/or control breaks.

  * columnar headers - their contents and
  generation criteria.

  * detail line(s) - their contents and generation
  criteria.

  * summary line(s) - their contents and generation
  criteria.

  * control break criteria.

  * control break footers - their content and
  generation criteria.

  * page break criteria includes full page and/or
  logical group break.
* identification of logical grouping criteria.

* page footer content.

* report groups a sequence of output lines that must appear on a single sheet.

* multiple report columns which require the reader to return to the top of the page to continue reading the report from the next column.

- **Types of Processing Actions used to form output values.** A single output line may use a combination of these actions. In fact, a single output value may use a combination of these actions.

  * reformatting of all fields - change order, spacing, use of commas, dollars decimal points, value size, removal of leading zeros, justification.

  * reformatting of selected fields.

  * decoding of input values into more readily understandable values.

  * encoding of input values usually into more cryptic forms.

  * combining fields from several sources either maintaining, or not, individual field values. The field names and the functions are programmer supplied.

  * combining fields of several records from a single source, either maintaining, or not maintaining, individual field values. Again, the I template traverses the source, but the user specifies the field names and how they are to be combined.

  * including program values.
1. Definition of the relationship(s) within a single source and/or among several sources. This would provide source code for identifying the relationship and bring related records together for processing.

2. Definition of the access statement(s) needed for each source that can include templates for local access to tables, files/databases.

**Inessential Dependent**

1. Concurrently Generated Reports - The simultaneous production of report lines for two or more separate reports.

2. Different Report Types - Two or more reports are called different, if when both are produced using identical sources with identical source content, and the resultant reports are not identical. Different report types occur when two reports differ in report category, format, record selection criteria.

In practice, different reports occur in two separate situations 1) a single program contains code which is capable of generating two or more different reports once each time the program is executed, or 2) a single program that generates two or more reports but not concurrently. The first situation usually occurs when the different reports have a significant amount of code in common which can be shared. The code that differs among the reports is controlled by parameter settings tested by control code. This first situation would be better handled using the template system which allows for variations to generate separate programs thus simplifying the content of the programs.

The second situation is a candidate for a linear sequence of two report generation programs that perhaps pass a file or table between them.

3. report title page contents including: requester's identification, date, billing and routing information.

4. report footer page contents including: file summary data,
Inessential Independent

1. Selection of source records to be processed - application of a criteria immediately upon accessing a source to determine if the record accessed is to be used or ignored. If it is to be ignored, an immediate access to the same source is made until a record meeting the selection criteria is retrieved.
D.9 Template Definition for: (Fabricate - Y)

PURPOSE

(Fabricate) manipulates source(s) data to produce output data in a new form, in a new format or organized according to new relationships. Unlike (Update) the output is not a newer version of one of the inputs and, although similar to (REPORT) in some respects, it does not contain the report semantics for dealing with reports except in the case of specific reports possibly produced during the fabrication process.

COMPOSITION - I, X

I - provides for traversal of the input source(s).

X - provides for generation of many possible reports.

SEMANTICS AND CLASSIFICATION

Essential Dependent

1. Disposition of fabricated records - at least one structure - file, database, or table must be created for holding fabricated records. These records must not have a format identical to any one of the sources which differentiates (Fabricate) from (Select), (Generalized Traversal), and (Sort). The disposition output must not have report semantics such as page breaks, titles and footers to differentiate it from (Report). Disposition is also optionally available for source records which are not selected.

2. Processing Actions - four processing actions are defined the last three of which can be used in combination: Ignore, Copy, Report and Manipulate. (Fabricate) has to determine when an action is needed and provide a
processing branch for each action.

3. Determining when to execute a processing action - this could be *fixed* for all selected records or could be *dependent* on a criteria such as record type, key value, position validity, presence of related records from other sources, or other procedurally determined criteria.

4. Manipulation action types - is included as an essential semantic because, if only ignore, report, and/or copy actions appear, then *(Fabricate)* is not needed as a separate template. The action types are: copy all fields as is, reformat all or selected fields, encode fields, decode fields, combine fields from several sources with or without loss of individual field identity, combine fields from a single source with or without loss of individual field identity, expand a single input into multiple outputs, and insert program values into the output.

**Essential Independent**

1. Traversal of at least one source is provided by the semantics obtained from the *(Generalized Traversal)* template.

**Inessential Independent**

1. Selection of source records to be considered for fabrication.

2. Reporting for one or several of the following purposes: error identification, source detail, source summary, or source detail and summary, output detail, output summary, or output detail and summary, or audit trail.
D.10 Template Definition for: (Table Traversal - H)

PURPOSE

This template traverses multi-level table(s) containing an arbitrarily complex structure at each level in the table.

COMPOSITION – I

This template contains a subset of the semantics present in the (Generalized Traversal) template.

SEMANTICS AND CLASSIFICATION

The semantics must allow for describing the following traversals: specific occurrences of data items, all occurrences of a data item, all occurrences of a level of a table possibly nested up to three levels deep, and a sequence of table-levels.

Essential Independent

1. Description of the structure of the table to be traversed which may not be the same as the definition of the table. This description must allow for: definition of table elements, nested table levels, and sequences of table-levels.

2. Element identity – identification of the table elements to be traversed.

3. Identification of process branch locations no semantic limitations on the type of processing to be accomplished.
D.II Template Definition for: (Table Initialization Using Fixed Values - A)

PURPOSE

This template initializes/reinitializes a table or portion of a table using fixed predefined values. The initialization is accomplished by traversing the table and moving element sized data into the table.

COMPOSITION - H

This template relies on the (Table Traversal - H) template to provide the needed traversal(s).

SEMANTICS AND CHARACTERISTICS

Essential Dependent

1. Definition of the element sized initialization data.
2. Relationship between the initialization data and the elements of the table.
3. Process actions that place data into table elements.

Essential Independent

1. The structure of the table to be traversed which is supplied by the H template.
2. Element identity supplied by the H template.
3. Process branch location supplied by the H template to contain the initialization steps.
D.12 Template Definition for: (Sequential Table Initialization Using a Sequential File - B)

PURPOSE

This template initializes/reinitializes a table or portion of a table by sequentially placing records from a sequential file into elements of the table.

COMPOSITION - H, I

The (Table Traversal - H) template semantics are used to traverse the table and the (Generalized Traversal - I) template semantics, restricted to a single sequential source are used to describe the file traversal, as well as the relationship between the elements of the table and the records of the file.

SEMANTICS AND CHARACTERISTICS

Essential Dependent

1. Process action semantics restricted to placement of file data into table elements.

Essential Independent

1. (Generalized Traversal) semantics are restricted to a single source using sequential access.

2. (Table Traversal) semantics are unrestricted with the exception of process actions which now are for the specific purpose of moving data into table elements.
D.13 Template Definition for: (Table Initialization by Directly Placing Records From a Sequential Source - J)

PURPOSE

A sequential source of records exists that contains data to be placed into a table. The identity of the table element to receive the data is also contained in the record or can be determined by a simple processing of record data. The template can be used to initialize or reinitialize table elements. A table thus initialized allows for direct retrieval of table elements during later processing.

COMPOSITION - I,H

The (Generalized Traversal - I) template is used to describe the needed file traversal and process branch locations for moving data into the table. The (Table Traversal - H) template provides table definition semantics.

SEMANTICS AND CHARACTERISTICS

Essential Dependent

1. Process Action restricted to placement of data into table elements.

2. Identification of table element to receive the data.
Essential Independent

1. Generalized Traversal semantics restricted to a single source which uses sequential access.

2. Table definition semantics from the (Table Traversal) template.
D.14 Template Definition for: (Indexed Linear Search of a Table - F)

PURPOSE

A table or level of a table is searched using the Cobol SEARCH verb to find the any number of elements meeting a condition. Although process locations are identified, no semantic restrictions are placed on the processes to be performed.

COMPOSITION - H

Because of the table traversal built into the Cobol SEARCH verb, the H template is only needed if all elements of a table meeting a condition are required. The H template semantics cause the search to resume at a point in the table following the last element to successfully meet the search condition. The H template is also needed if all or several occurrences of a nested level of a table are to be searched. The H template is responsible for establishing outer table level indices in this case.

SEMANTICS AND CHARACTERISTICS

Essential Independent

1. Search criteria or condition
2. Identity of table level(s) to be searched
3. Number of items to be Search for
4. Found process action(s)
5. Not found process action(s)
6. Description of the Indexed table
D.15 Template Definition for: (Linear Search of a Table - G)

PURPOSE

A table or level(s) of a table are linearly searched to locate any number of elements meeting a condition. Locations are identified for process code.

COMPOSITION - H, C

The (Table Traversal - H) template provides the traversal needed during the search. The (Case - C) template can be used to evaluate several search conditions.

SEMANTICS AND CHARACTERISTICS

Essential Independent

1. Search criteria or condition
2. Identity of table levels to be searched
3. Description of the table to be searched which could be indexed or not
4. Number of items to be Search for
5. Found process action(s)
6. Not found process action(s)
D.16 Template Definition for: (Binary Search of a Table - P)

PURPOSE

A level of a table whose elements are arranged in ascending or descending sequence are searched to find the first element meeting a condition. Locations for process code are identified.

COMPOSITION

The Cobol SEARCH ALL verb is used to implement the template when the table is indexed.

SEMANTICS AND CHARACTERISTICS

Essential Dependent

1. **Description** of an ordered table

2. Identification of the table item used to order the table and whether it's ascending or descending order.

Essential Independent

1. **Search Criteria or condition**

2. **Found** process action(s)

3. **Not found** process action(s)
Appendix E.

Case Study Semantics Applied to the PDM and Hypothesis Template Sets

In order to better understand the limitations present in both the PDM Model Programs and the Hypothesis Template Set all the templates from both sets are rated using the semantics developed for application programs in the case study. A scrutiny of the tables in this appendix will reveal the semantics defined via the case study that were not present in the PDM and Hypothesis Template Sets.

E.1 Generalized Traversal Semantics Applied to Appropriate PDM Model Templates

PDM MODEL TEMPLATE IDENTITY

The numbers correspond to the following PDM Model Templates:

1. Extract Table File
2. Convert Data File
3. Edit Transaction File
4. Update Master File
5. Edit Sort File
6. Create Detail/Summary Report
**DATA SUMMARY**

<table>
<thead>
<tr>
<th>PIM Template Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Total Nr Sources</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>2. Sources by Type</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Database</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>3. Nr. of</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rec Types</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Group Types</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Key</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Delimiter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. Deepest Group Nesting</strong></td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>5. Fixed Seq. Max. Length</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>6. Variable Seq. Max. Length</strong></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>7. Selectivity By</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Rec Type</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Key</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
E.2 Generalized Traversal Semantics Applied to Appropriate Hypothesis Set Templates

**HYPOTHESIS SET TEMPLATE IDENTITY**

The numbers correspond to the following Hypothesis Templates:

1. Match Sequential
2. Match Random
3. Iteration

**DATA SUMMARY**

<table>
<thead>
<tr>
<th>Hypothesis Templates</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Total Nr Sources</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sources</td>
<td>2</td>
<td>2-M</td>
<td>1</td>
</tr>
<tr>
<td>2. Sources By Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File</td>
<td>2</td>
<td>1</td>
<td>2-M</td>
</tr>
<tr>
<td>Database</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3. Nr. of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rec Types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group Types</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>0</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>Delimiter</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4. Deepest Group</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nesting</td>
<td>N/A</td>
<td>N/A</td>
<td>M</td>
</tr>
<tr>
<td>5. Fixed Seq.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. Length</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6. Variable Seq.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max. Length</td>
<td>2</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>7. Selectivity By</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Rec Type</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Key</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>
### DATA SUMMARY

<table>
<thead>
<tr>
<th>PDM Model Template</th>
<th>Hypothesis Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert-Data-File</td>
<td>None</td>
</tr>
</tbody>
</table>

#### Files/Databases

1. Nr Sources by Type
   - Files: 1
   - Database: 0
   - Tables: 0

2. Source Access
   - Sequential: 1
   - Random: 0

3. Nr. Rec. Types: 1

4. Deepest Group Nesting: 0

5. Source:Source Relationship: N/A

6. Nr. Reports: 0

7. Report Purposes: None

8. Nr. Outputs by Type
   - Files: 1
   - Database: 0
   - Tables: 0

9. Nr. Rec. Types in Outputs: 1

10. Deepest Group Nesting in Output: 0

11. Output use
    - Fabricated: Y
    - Not Selected: N
12. Source:Output Relationship 1:1

13. Source Selectivity

   All Y
   Determined by N

B. Fabricate Actions

1. For Selected Source Recs.
   - Ignore N
   - Copy Y
   - Report N
   - Manipulate Y

2. Action Determined by
   Fixed Y
   Determined by N

C. Manipulation Actions

1. Copy All Fields Y

2. Reformat
   All Fields Y
   Selected Fields Y

3. Encode N

4. Decode N

5. Combine Several Sources N

6. Combine Recs. from Single Source N

7. Expand Single Source N

8. Include Pgm. Fields N
E.4 Report Generation Semantics Applied to Appropriate PDM and Hypothesis Set Templates

PDM MODEL AND HYPOTHESIS SET TEMPLATE IDENTITY

1. PDM Model template - Create Detail.
2. PDM Model template - Create Summary.
3. PDM Model template - Create Detail/Summary.
### DATA SUMMARY

#### A. FILES/DATABASE/DATA STRUCTURES

<table>
<thead>
<tr>
<th>PDM TEMPLATES</th>
<th>HYPOTHESIS TEMPLATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 A 5 6</td>
<td>4 5 6</td>
</tr>
</tbody>
</table>

1. **Number of Sources by Type**

<table>
<thead>
<tr>
<th>External</th>
<th>Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1</td>
<td>0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

2. **External Source Access**

<table>
<thead>
<tr>
<th>Record types</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
<th>Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>External</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
</tbody>
</table>

3. **External Source Key Group Levels**

   | N/A 2 2          |
   | many many       |

4. **External Source Selectivity**

   | ALL | Y | Y | Y | Y | Y | Y |

   Determined by
   - **Record Type**
     - N N N N N N N
   - **Key value**
     - N N N N N N N
   - **Position**
     - N N N N N N N
   - **Validity**
     - N N N N N N N
   - **Other source**
     - N N N N N N N
   - **Procedure**
     - N N N Y Y Y

5. **Internal Source Number Levels Access**

   | 0 0 0 0 0 0 |

6. **Source Relationship Defined by**

   | Key group | N/A Y Y Y Y Y |
   | Position | N N N N N N |
   | Delimiter | N N N N N N |

7. **Number of Concurrent Reports**

   | Reports | 1 1 1 1 1 1 |
   | Detail  | 1 N/A N/A 1 N/A N/A |
   | Summary | N/A 1 N/A 1 N/A N/A |
   | Both    | N/A N/A 1 N/A N/A 1 |
### PDM TEMPLATES

<table>
<thead>
<tr>
<th>Types</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detail</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Summary</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>Both</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>1</td>
</tr>
</tbody>
</table>

### REPORT FORMATS

1. **Global Concerns**
   - None
   - Report Heading
     - Y
   - Report Footing
     - Y

2. **Page Heading**
   - None
   - Pg #
     - Y
   - Pgm ID
     - Y
   - Run Date
     - Y
   - Data Date
     - Y
   - When Generated
     - .full page
       - Y
     - .control break
       - N/A
     - .group identity
       - N/A
     - .title line
       - Y
     - .every detail line
       - Y
     - .detail line only when changed
       - Y

3. **Footings Used**
   - None
   - Page
     - Y
   - Group
     - N/A
     - Y
     - Y
### 4. Types of Output Actions

<table>
<thead>
<tr>
<th>PDM TEMPLATES</th>
<th>HYPOGESIS TEMPLATES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  2  3</td>
<td>4  5  6</td>
</tr>
</tbody>
</table>

#### Formatted
- Copy of all Fields (See A.3)
  - Y  Y  Y  Y  Y  Y
- Copy of selected fields
  - N  N  N  N  N  N
- Encoding
  - N  N  N  N  N  N
- Decoding
  - N  N  N  N  N  N

#### Combine
- Several Sources
  - N  N  N  N  N  N
- Maintain Field Integrity
- Loss of Field Integrity

#### Combine Recs. From Single Source
- N/A  Y  Y  N/A  Y  Y
- Maintain Field Integrity
  - N  N  N  N  N
- Loss of Field Integrity
  - Y  Y  Y  Y

#### Expand single source
- N  N/A  N  N  N/A  N

#### Include program fields/not title
- N  N  N  N  N  N

### 5. Use of Report Groups
- N  N  N  N  N  N

### 6. Multiple Homogeneous Column Output
- N  N  N  N  N  N
E.5 Table Semantics Applied to Appropriate PDM and Hypothesis Templates

**DATA SUMMARY**

<table>
<thead>
<tr>
<th>PDM MODEL TEMPLATES</th>
<th>HYPOTHESIS SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort Table File</td>
<td>N/A</td>
</tr>
<tr>
<td>Extract Table File</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1. NR TABLES</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Levels</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. INIT by</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Fixed Null Values</td>
<td></td>
</tr>
<tr>
<td>Redefines</td>
<td></td>
</tr>
<tr>
<td>Table Level</td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td></td>
</tr>
<tr>
<td>A template</td>
<td></td>
</tr>
<tr>
<td>Ripple Move</td>
<td></td>
</tr>
<tr>
<td>-Unique Data</td>
<td></td>
</tr>
</tbody>
</table>
|   Redefines | Y
|   Table Level | |
|   Move | |
|   J template | |
|   B template | |
|   Other | Y
| -When File Used | |
|   Rows Selected By | |
|     Type | Y
|     Key | Y
|   Fields Used | |
|   All | Y
|   Selected | Y
| -Not Initialized | |

<table>
<thead>
<tr>
<th>4. Reused</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5. Reused How</th>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDM MODEL TEMPLATES</td>
<td>HYPOTHESIS SET</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td>Sort Table File</td>
<td>Extract Table File</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6. Reuse When
- Control Break
- New Rec
- New Rec and EOF

7. Use Placing Data Into Table
- N/A
- Insert Unique Data Into Element
- Augment Current Element Data
- Element Identity
  - Direct
  - Next
  - Search
  - F template
  - G template

8. Use Accessing Values From
- Entire Table
- Element
  - H template
  - Direct
  - Search
  - F template
  - G template
  - Bubble Sort

9. Purpose
- Decode
- Set membership
- Char Access
- Save Images
- Accumulate Values
- Summary Flag
- Supply Update Values
- Unspecified
- Order Table

<table>
<thead>
<tr>
<th>N/A</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
E.6 Update Semantics Applied to Appropriate PDM and Hypothesis Templates

**DATA SUMMARY**

A. Files/Databases/Data Structures

<table>
<thead>
<tr>
<th>Field</th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPDATE-MASTER-FILE</strong></td>
<td></td>
<td><strong>UPDATE</strong></td>
</tr>
<tr>
<td><strong>1. Transaction Source</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>1</td>
<td>0-2</td>
</tr>
<tr>
<td>Access</td>
<td>seq</td>
<td>seq</td>
</tr>
<tr>
<td>NR Rec types</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>NR Key Groups</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>Sorted in Pgm.</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Avail. Trans.</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>Master Identity</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Action</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td><strong>2. Transaction Output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Partitioning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fabricated</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>3. Master Source</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Access</td>
<td>seq/ran</td>
<td>seq/ran</td>
</tr>
<tr>
<td>NR Rec. Types</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>NR Key Groups</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>Levels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Available</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>Master recs</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4. Master Out</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-place Update</td>
<td>N</td>
<td>Y/N</td>
</tr>
<tr>
<td>Access</td>
<td>seq/ran</td>
<td>seq/ran</td>
</tr>
<tr>
<td>Rec. Types</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>Group Types</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>5. LOG Files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Master Before</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>-Master After</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>-Trans.</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
6. Reports
Number 1 1
Kind
- error  Y  Y
- Transaction  Detail  N  N
- Transaction  Summary  N  N
- All Master  Detail  N  N
- All Master  Summary  N  N
- Changed Master  Detail  N  N
- Changed Master  Summary  N  N
- Action Details  N  N

B. Transaction Details

1. Actions
Number 3 3
Delete  Y  Y
Add  Y  Y
Change  Y  Y
Combination  N  N

2. Action Type Indicator
Field in Record  Y  Y
Group Delimiter  N  N
Fixed Assumed  N  Y
Parameterized  N  N

3. Master/Transaction Relation
1:N  Y  Y
- Grouped By
  Key  Y  Y
  Delimiter  N  N
- Local Loops  N  Y
1:1  N  N
M:1  N  N
M:N  N  N
### PDM Model Update-Master-File vs. Hypothesis Set Update

#### 4. Comparison Key

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Single Field</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Multi-field</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>-construct key</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>-individual field test</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Enumerated</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Procedurally Determined</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

#### 5. Delete Action

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write With Flag</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>No Write</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Physically Removed</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

#### 6. Change Action

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>Not specified</td>
<td></td>
</tr>
<tr>
<td>-Replacement</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>-Augmentation</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Positional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Replacement</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>-Augmentation</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

#### 7. Add Action

- No data collected other than that reported in B.1

#### 8. Error Testing

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Non-existing</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Add Existing</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Change Non-existing</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Integrated Validate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasks</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Out-of-date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transaction</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

#### 9. Ordering of Transactions By

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Keys</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Order Submitted</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Action Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record Type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 10. Transaction Sequences

<table>
<thead>
<tr>
<th></th>
<th>PDM Model</th>
<th>Hypothesis Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Legal</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Restrictions</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Appendix F.

Definition of Attribute Grammar Forms

The definitions given in this appendix are taken from Kuo [Kuo83c]. An extensive development and description of these definitions is presented by Soni [Soni83a].

**Grammar Form Definition**

A context-free grammar \( G = (V, N, S, P) \) is a four-tuple, where

1. \( V \) is a finite set of vocabulary,
2. \( N \) is a finite set of nonterminal vocabulary and \( N \) is contained in \( V \),
3. \( S \) is the starting symbol contained in \( N \), and
4. \( P \) is a finite set of production rules of the form \( X \rightarrow a \), where \( X \) is in \( N \) and \( a \) is in \( V^* \).

Since \( G \) has a finite vocabulary it is, by itself, inadequate for modeling a program derivation tree whose nodes may be labeled by strings belonging to an infinite set — the design language. Hence the context-free grammar is augmented as shown below. A grammar form is composed of a **concept grammar** that is a "pattern" for constructing other grammars which are called the **interpretation grammars**, that have similar form. The interpretation grammars are

404
supplied by the programmer by responding to prompts of the concept
grammar symbols in some language, or by the template definer when
semantics represented in existing template definitions are
transformed to yield a new template definition.

More formally, a (context-free) grammar form \( G \) is a four-tuple
\((G, M, V, S)\) where

1. \( G = (V, N, S, P) \) is a context-free grammar called the
   concept grammar,
2. \( M \) is an infinite set of substitutions, and
3. \( V \) and \( S \) are infinite vocabularies with \( S \) in \( V \).

For each \( u \) in \( M \),

1. \( u(a) \) is a finite set in \( S^* \), \( a \) in \( V-N \),
2. \( u(X) \) is a finite set in \( V - S \), \( X \) in \( N \), and
3. if \( X, Y \) are in \( N \) and \( X \neq Y \), then \( u(X) \) and \( u(Y) \) are
   disjoint.

The convention of referring to a grammar form by its underlying
concept grammar \( G \), while \( M \), \( V \) and \( S \) are understood will be used.

An interpretation grammar \( G(I) = (V(I), N(I), S(I), P(I)) \) of \( G \)
is a grammar such that

1. \( u(V) = V(I) \),
2. \( u(N) = N(I) \),
3. \( u(V-N) = V(I) - N(I) \),
4. \( P(I) \) is contained in \( \{ u(X) \rightarrow u(a) \mid X \rightarrow a \) in \( P \} \), and
5. \( S(I) \) is contained in \( u(S) \).
ATTRIBUTE GRAMMAR FORMS

As in the definition of a grammar form which is derived primarily based on its underlying concept grammar, an attribute grammar form is defined based on its underlying concept attribute grammar. Since the notion of attribute grammars is not as well known as context-free grammars, first a definition [Knut74, Raih77] of the attribute grammars will be given, followed by a definition of attribute grammar forms.

Definition of Attribute Grammars

An attribute grammar AG is a four-tuple AG = {G, A, F, R}, where

1. G = {V, N, S, P} is a context-free grammar,
2. A = {Syn, Inh, Ran, Val} is a specification of attributes,
3. F is an attribute associator function for G and A, and
4. R is a set of collections of semantic rules.

A specification of attributes A for a grammar G is a four-tuple A = {Syn, Inh, Ran, Val}, where

1. Syn is a finite set of synthesized attributes, disjoint from Val.
2. Inh is a finite set of inherited attributes, disjoint from Val and Syn.
3. Ran is a collection of sets of allowed attribute values.
4. Val is a mapping from Syn and Inh to Ran.
The set Syn \( U \) Inh is denoted by \( A \) and called the set of attributes. That is, \( A \) includes both the synthesized and inherited attributes. For each attribute \( a \) in \( A \), \( \text{Val}(a) \) specifies the set of values in \( \text{Ran} \) that any instance of attribute \( a \) may assume.

To each symbol \( x \) in \( V \), is associated a finite set \( F(x) \) of attributes. This set \( F(x) \) is partitioned into two disjoint sets, the synthesized attributes \( F_s(x) \) and the inherited attributes \( F_i(x) \). It is also required that \( F_i(S) \) to be empty (that is, the starting symbol \( S \) has no inherited attributes) and \( F_s(x) \) to be empty if \( x \) is in \( V-N \) (that is, the terminal symbols have no inherited attributes).

The mapping, from \( V \) to the power set of \( A \), which satisfies the above conditions is called the attribute associator for \( G \) and \( A \).

Let

\[
P: x_0 := x_1 \ x_2 \ldots \ x_n
\]

be a production of \( G \). \( P \) has an attribute occurrence \((a,j)\) if \( a \) is an attribute in \( F(x_j) \) and \( 0 < j \leq n \). That is, \( a \) is an attribute belonging to the set of attributes associated with the \( j \)-th symbol in the production \( P \). A semantic rule for an attribute occurrence \((a,j)\) in production \( P \) is a function

\[
r: \text{Val}(b_1) \times \text{Val}(b_2) \times \ldots \times \text{Val}(b_n) \rightarrow \text{Val}(a)
\]

where each \((b_i, i)\) is an attribute occurrence of \( P \). In other words, each semantic rule maps values of certain attributes of \( x_1 \), \( x_2 \), \ldots, \( x_n \) into the value of some attribute of \( x_j \). It is required
that each attribute occurrence of a production $P$ must have one and only one semantic rule. Each production thus has a collection of semantic rules for all of its attribute occurrences. The set of collections of semantic rules for all productions is denoted by $R$.

**Attribute Grammar Forms**

Modelling the formation of template definitions by transforming and combining existing template definitions, and representing the instantiation process wherein users by responding to prompts transform a template definition into a skeletal program requires all if not more of the capabilities present in attribute grammars and grammar forms. This led Soni [Soni83a] to merge the features of the two by replacing the context-free grammar usually associated with grammar forms with an attribute grammar. For an extensive definition and discussion of the attribute grammar form model see [Soni83a].
Appendix C.

Forms Used During Program Centered Analysis

PROGRAM STUDY FORM 1

A. PROGRAM IDENTITY

1. Program Name
   [Taken from the PROGRAM-ID of the IDENTIFICATION DIVISION of the source listing.]

2. Program Source
   [Identifies the company and the company contact that supplied the program.]

3. Date Collected
   [Date the source listing and documentation was obtained.]

4. Evaluation Time
   [The amount of time expended in completing these three Program Study Forms for a program. This begins with the initial reading of documentation and ends when all three Forms are complete. It does not include any of the Pattern Centered data summarization.]

B. GENERAL DESCRIPTION of the PROGRAM

   [A description of the function and purpose of the program, determined by manual analysis of the source code and any supplied documentation.]

C. FILE/DATABASE/INTERNAL DATA STRUCTURE DIAGRAM

   [A diagram using standard device symbols which identifies structures, etc., by internal names. Indicates the access direction(s) (i.e., input, output, output then input, input and output). This was invaluable in gaining an understanding of the application, program control structure and patterns used.]
D. FILE/DATABASE/INTERNAL DATA STRUCTURE ANNOTATIONS

[For each of the entities named in part C, a brief description of the purpose, contents and organization is given.]

E. JACKSON DIAGRAMS

[For each of the files/database(s) identified in C, a Jackson structure diagram was constructed. These diagrams would be appropriate for the merging phase of the Jackson methodology. This merging of individual diagrams into a single program structure diagram was not attempted.]

F. ALGORITHM NOTES

[Comments on unique aspects of the application or interesting implementation or style techniques encountered in the program.]
PROGRAM STUDY FORM 2

A. PROGRAM NAME
   [As explained in Form 1]

B. TEMPLATES and the ORIGINAL PROGRAM

   1. List of templates which could be identified.
      [The names of templates which could be identified as being present in the original program source without concern for nesting or sequencing of the templates.]

   2. Template Interaction
      [A hierarchical diagram with templates as the nodes depicting the nesting of templates present in the original program.]

C. TEMPLATE-BASED METHODOLOGY SOLUTION DIAGRAM

   [Using the understanding of the application requirements and the existing file environment, the template-based methodology was applied to produce a solution. The solution frequently included additional temporary sequential files which formed interfaces between templates.]

D. TEMPLATE-BASED SOLUTION FILE ANNOTATIONS

   [Description of additional files included in the template-based solution beyond those present in the original file environment.]

E. DISCUSSION of the TEMPLATE-BASED SOLUTION

   [Any shortcomings of the solution, unique requirements or rational for the solution generated are described.]
A. PROGRAM NAME

B. FILE/DATABASE/DATA STRUCTURE Counts

1. From the Original Program
   [Each of the files and databases used in the program are counted by access method and direction of data flow (e.g., Sequential Input). Reports are Sequential Output files but were identified as a separate category. Frequently files existed which was created by the program and then input back into the program (i.e., SEQUENTIAL OUTPUT then INPUT). Internal data structures (tables) which played a significant role in the program were also identified and counted.]

2. Additional Structures Introduced with the Template Solution.
   [Counted as described in B.1 above.]

3. A Comparative Analysis of I/O Requirements
   [Because of the concern over additional I/O generated by the template based solution as compared to the original program solution, a formula was developed to identify the difference in I/O requirements. The formula was necessarily based on application dependent variables such as the number of valid records in the transaction file.]

C. ORIGINAL PROGRAM DEMOGRAPHICS

1. Number of Source Lines
   [As a measure of program size, the number of source lines as indicated by the compiler's line number count was recorded.]

2. Procedure Division Source Lines
   [The number of source lines in the procedure division was determined by subtracting item C.1 above from the compiler line number of the "PROCEDURE DIVISION" statement.]