Augmenting the relational model with conceptual graphs

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

Additional Information:

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

Metadata Record: [https://dspace.lboro.ac.uk/2134/10426](https://dspace.lboro.ac.uk/2134/10426)

Publisher: © Brian Andrew Bowen

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

**Attribution-NonCommercial-NoDerivs 2.5**

You are free:
- to copy, distribute, display, and perform the work

**Under the following conditions:**

**By:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Noncommercial.** You may not use this work for commercial purposes.

**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
<table>
<thead>
<tr>
<th>AUTHOR/FILING TITLE</th>
<th>Bowen, B. A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESSION/COPY NO.</td>
<td>040129489</td>
</tr>
<tr>
<td>VOL. NO.</td>
<td></td>
</tr>
<tr>
<td>CLASS MARK</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date Due</th>
<th>Loan Copy Expired</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Dec 1996</td>
<td>29 Jun 1999</td>
</tr>
<tr>
<td>Loan 3 Wks. + 3 Unl.</td>
<td></td>
</tr>
<tr>
<td>UW 62809</td>
<td></td>
</tr>
<tr>
<td>13 Jun 1997</td>
<td>14 Aug 2000</td>
</tr>
<tr>
<td>3 Oct 1997</td>
<td>08 Jan 2001</td>
</tr>
<tr>
<td>2 Oct 1998</td>
<td></td>
</tr>
</tbody>
</table>
Augmenting the Relational Model
With Conceptual Graphs

by

Brian Andrew Bowen

A Doctoral Thesis

Submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy of the
Loughborough University of Technology

October 1995

© Brian Andrew Bowen 1995
Abstract

While the relational model for data storage is sufficient for the modelling and manipulation of a large number of application domains, a growing class of application domains are either difficult or impossible for the relational model to deal with efficiently. The realisation of this fact has led to a proliferation of data models that attempt to increase the complexity and semantic capture of the domains that they can model - the development of object-oriented databases and the various semantic data models are a result of this.

The idea of using logic to define, manipulate and constrain data has given rise to large numbers of systems that interface - not always successfully - a database system and a logic processing system. Most such systems are based on Prolog or its derivations.

This thesis describes the development and use of an object-oriented and semantically rich form of logic - conceptual graph theory - as a system for the definition, manipulation, and constraint of data. It describes a theoretical correspondence between conceptual graph theory and the relational model, and proceeds to develop an augmented, hybrid theory that is formally more expressive and as rigorous as those languages based on the relational algebra or calculus.

This thesis also describes the design and implementation of a hybrid relational database - conceptual graph system, that has a cleaner and more principled system of semantic capture than other (for example, Prolog-based) systems, and that is also adaptive in nature - it automatically modifies its underlying storage structures in accordance with modifications made to the structures of the application domain over time. This completely shields the user from any responsibility for database design and maintenance, and so the user need only be concerned with application domain knowledge. Although the implementation described is incomplete, it can be extended to produce a deductive, object-oriented database system based on conceptual graphs.
Acknowledgements

I am deeply indebted to my project supervisor, Pavel Kocura, for four years of (generally) amicable association and argument, as well as a myriad of useful insights. I must also express my thanks to Dr. John Heaton and Graham Hill of the Conceptual Graphs Development Group at Loughborough University, for many valuable and fruitful discussions about the limitations and development of conceptual graph theory.

Acknowledgement should also be made of Jon Blakeley, who initiated some useful developments in the coding of graph assertions, and of Alfred Chan, for some interesting ideas on implementational extensions to the work contained in this thesis.

Finally, this acknowledgment would be incomplete if I did not thank my parents for their love, and for their financial support during the later stages of my research.
Contents

Abstract i
Acknowledgements ii
Certificate of Originality iii
Contents iv
Appendices xiii
List of Figures xiii

1. Introduction 1

1.1 Advantages of the Relational Model Over Earlier Models 2
1.1.1 Importance of Logical Data Organisation 2
1.1.2 Conceptual Simplicity 3
1.1.3 Declarativeness 3

1.2 Disadvantages of the Relational Model 4
1.2.1 Difficulties of Modelling Complex Structures 4
1.2.2 Limitations of Relational Manipulation Languages 5
1.2.2.1 Logical Basis of Limitations 5
1.2.2.2 Lack of Transitive Closure of a Binary Relation 6
1.2.2.3 General Lack of Deductive Retrieval 7
1.2.2.4 Lack of Reasoning with Metadata 7
1.2.2.5 Lack of Formal Computational Basis 8

1.2.3 Semantic Limitations of the Relational Model 8
1.2.3.1 Representation of Negative Information 8
1.2.3.2 Representation of Disjunctive and Indefinite Data 8
1.2.3.3 Representation of Hierarchical Information and Encapsulation 9
1.2.3.4 Problem of Domain Join Mismatch 9
1.2.3.5 Program-Level Semantics are Vulnerable to Abuse or Error 10
1.2.3.6 Summary of Semantic Capture in the Relational Model 10

1.3 Extending the Relational Model 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.1 Some Obvious Extensions</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2 Some Less Obvious Extensions</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2.1 Declarativeness Again</td>
<td>11</td>
</tr>
<tr>
<td>1.3.2.2 High-Level Representation</td>
<td>13</td>
</tr>
<tr>
<td>1.3.3 Appearance of Postrelational Systems</td>
<td>13</td>
</tr>
<tr>
<td>1.3.4 Object-Oriented Data Models</td>
<td>14</td>
</tr>
<tr>
<td>1.3.4.1 Basic Object-Oriented Philosophy</td>
<td>14</td>
</tr>
<tr>
<td>1.3.4.2 Advantages of Object-Orientation</td>
<td>15</td>
</tr>
<tr>
<td>1.3.4.3 Disadvantages of Object-Orientation</td>
<td>15</td>
</tr>
<tr>
<td>1.3.4.4 Fuzziness of Object-Orientation</td>
<td>16</td>
</tr>
<tr>
<td>1.3.4.5 Common Ground Between the Relational Model and Object-Orientation</td>
<td>17</td>
</tr>
<tr>
<td>1.3.5 Semantic Data Models</td>
<td>18</td>
</tr>
<tr>
<td>1.3.4.1 Basic Semantic Data Model Philosophy</td>
<td>19</td>
</tr>
<tr>
<td>1.3.4.2 Value of Increased Semantics</td>
<td>20</td>
</tr>
<tr>
<td>1.3.4.3 Why Don't Semantic Data Models Rule the (Database) World?</td>
<td>20</td>
</tr>
<tr>
<td>1.4 Logic and the Relational Model</td>
<td>21</td>
</tr>
<tr>
<td>1.4.1 A Model-Theoretical View</td>
<td>22</td>
</tr>
<tr>
<td>1.4.2 A Proof-Theoretical View</td>
<td>22</td>
</tr>
<tr>
<td>1.4.3 Relevance of Proof-Theoretical View</td>
<td>23</td>
</tr>
<tr>
<td>1.4.4 'Hybridisation': Artificial Intelligence and Relational Databases</td>
<td>23</td>
</tr>
<tr>
<td>1.4.4.1 Potential Benefits of Hybridisation</td>
<td>24</td>
</tr>
<tr>
<td>1.4.4.2 Separation of Data and Semantics</td>
<td>25</td>
</tr>
<tr>
<td>1.4.5 Approaches to Hybridisation</td>
<td>26</td>
</tr>
<tr>
<td>1.4.5.1 Loose-Coupled Approach</td>
<td>27</td>
</tr>
<tr>
<td>1.4.5.2 Tight-Coupled Approach</td>
<td>28</td>
</tr>
<tr>
<td>1.4.5.3 Comparison of Loose-Couples and Tight-Couples</td>
<td>29</td>
</tr>
<tr>
<td>1.4.6 Graphical Representations of Logic</td>
<td>30</td>
</tr>
<tr>
<td>1.4.6.1 Advantages of a Graphical Notation</td>
<td>31</td>
</tr>
<tr>
<td>1.4.7 Conceptual Graph Theory</td>
<td>31</td>
</tr>
<tr>
<td>1.4.7.1 Subsumption of Other Relational Graphical Notations by Conceptual Graph Theory</td>
<td>32</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.4.7.2 Conceptual Graph Theory as a Semantic Data Model with a Manipulative Component</td>
<td>32</td>
</tr>
<tr>
<td>1.4.8 Use of Conceptual Graphs as a Hybrid Manipulation Language</td>
<td>33</td>
</tr>
<tr>
<td>1.5 Summary</td>
<td>34</td>
</tr>
<tr>
<td>1.6 Research Aims</td>
<td>34</td>
</tr>
<tr>
<td>1.6.1 Primary Aim: Conceptual Graphs as a Conceptual Schema Definition Language</td>
<td>35</td>
</tr>
<tr>
<td>1.6.2 Aim #2: Correspondence Between Conceptual Graphs and the Relational Model</td>
<td>35</td>
</tr>
<tr>
<td>1.6.3 Aim #3: Use of Extended Conceptual Graphs in a Hybrid System</td>
<td>35</td>
</tr>
<tr>
<td>1.6.4 Aim #4: Increased Declarativeness of a Hybrid System</td>
<td>36</td>
</tr>
<tr>
<td>1.6.5 Aim #5: Use of Existing Technology in Hybrid System Construction</td>
<td>36</td>
</tr>
<tr>
<td>1.6.6 Aim #6: Investigation of Legacy Database Reuse in a Hybrid System</td>
<td>36</td>
</tr>
<tr>
<td>1.7 Organisation of Thesis</td>
<td>37</td>
</tr>
<tr>
<td>2. Conceptual Graphs and the Relational Model</td>
<td>38</td>
</tr>
<tr>
<td>2.1 Chapter Objectives</td>
<td>38</td>
</tr>
<tr>
<td>2.2 Correspondence Between Relational Structures and Graph Structures</td>
<td>39</td>
</tr>
<tr>
<td>2.2.1 Formal Structural Description of the Relational Model</td>
<td>39</td>
</tr>
<tr>
<td>2.2.2 Mapping of Conceptual Graphs to Relational Structures</td>
<td>42</td>
</tr>
<tr>
<td>2.2.3 Similarity of Domain-Extended Conceptual Graphs to Hypergraphs</td>
<td>46</td>
</tr>
<tr>
<td>2.2.3.1 Datalog and Hypergraphs</td>
<td>46</td>
</tr>
<tr>
<td>2.2.4 Domain-Extended Conceptual Graphs as Window Functions</td>
<td>47</td>
</tr>
<tr>
<td>2.2.5 Advantages of a Universal Relation Approach</td>
<td>47</td>
</tr>
<tr>
<td>2.2.6 Null Information in the Relational Model and Conceptual Graphs</td>
<td>48</td>
</tr>
<tr>
<td>2.3 Correspondence Between Relational Operations and Graph Operations</td>
<td>52</td>
</tr>
<tr>
<td>2.3.1 Description of the Relational Algebra</td>
<td>52</td>
</tr>
<tr>
<td>2.3.2 Why the Relational Algebra is Limited: Problems of General-Purpose Languages</td>
<td>53</td>
</tr>
<tr>
<td>2.3.3 Formal Specification of the Tuple Relational Calculus</td>
<td>54</td>
</tr>
<tr>
<td>2.3.3.1 Examples of Tuple Relational Calculus Formulas</td>
<td>56</td>
</tr>
<tr>
<td>2.3.3.2 Requirements for Construction of Legal Tuple Relational Calculus Formulas</td>
<td>57</td>
</tr>
<tr>
<td>2.3.3.3 Formation of Tuple Relational Calculus Expressions</td>
<td>59</td>
</tr>
<tr>
<td>2.3.4 Conceptual Graphs and the Tuple Relational Calculus</td>
<td>60</td>
</tr>
<tr>
<td>2.3.4.1 Canonical Formation Operations and Domain Mappings</td>
<td>61</td>
</tr>
</tbody>
</table>
3.2.3 The Binary Relational Approach

3.2.3.1 Advantages of Binary Relational Approach

3.2.3.2 Disadvantages of Binary Relational Approach

3.2.4 TRISTARP and Related Advances

3.2.5 Comparison of N-ary and Binary Relational Approaches

3.2.6 Early Use and Rejection of Binary Relational Approach

3.2.6.1 Hierarchical Problems with Binary Relational Strategy

3.2.6.2 Encapsulation Problems with Binary Relational Strategy

3.2.6.3 Maximal Expansion in Binary Relational Strategy

3.2.6.4 Disadvantages of Maximal Expansion

3.2.7 Adoption of N-ary Relational Approach

3.2.7.1 Hierarchical Problems with Naive N-ary Relational Strategy

3.2.7.2 N-ary Relational Strategy and Some Hierarchical Assumptions

3.2.7.3 Hierarchy and Retrieval in an N-ary Relational Strategy

3.2.7.4 Impracticality of Non-Expansive N-ary Relational Strategy

3.2.8 Summary of Storage Issues

3.2.8.1 Suitability of N-ary Relational Approach for Pure Data Storage

3.2.8.2 'Recasting' of Primitive Conceptual Relations

3.2.8.3 Binary Relational Approach as a 'Repository'

3.3 Database Issues II: Adaptive Storage

3.3.1 Cardinality and Membership in Adaptive Storage

3.3.1.1 Storage for Primitive Relations

3.3.1.2 Specialisation of Default Cardinality and Membership

3.3.1.3 Comparison of Adaptive Information Against Previous Knowledge

3.3.1.4 Schema Restructuring: Addition, Modification and Deletion

3.3.1.5 Structure of Cardinality

3.3.1.6 Computation of Normalised Storage

3.3.1.7 Circumstances Under Which Adaptation Occurs

3.4 Conceptual Graph Issues I: Graph Structures in C-GRASS

3.4.1 Type Hierarchy

3.4.1.1 Two Types of Hierarchical Arc

3.4.1.2 Metatyping of Concepts
3.4.1.3 Deliberate Limitations of Metatyping
3.4.2 Relation Hierarchy
3.4.3 Type and Relation Definitions
3.4.4 Canonical Model
3.4.5 Application Graph
3.4.6 Function Definitions

3.5 Conceptual Graph Issues II: Graph Operations in C-GRASS
3.5.1 Basic Conceptual Graph Operations in C-GRASS
   3.5.1.1 Canonical Formation Rules in C-GRASS
   3.5.1.2 Join
   3.5.1.3 Projection
   3.5.1.4 Encapsulation Operations
   3.5.1.5 \(\phi_A\)
3.5.2 Macro Operations in C-GRASS
   3.5.2.1 Application Graph Assertion
   3.5.2.2 Canonical Model Assertion
   3.5.2.3 Lattice Assertion
   3.5.2.4 Type, Relation and Function Definition
   3.5.2.5 Inquiry
   3.5.2.6 Miscellaneous Operations - Display and Quit
   3.5.2.7 Future Functionality

3.6 Conceptual Graph Issues III: Computation in C-GRASS
   3.6.1 Aggregate Functions
   3.6.2 Arithmetic Functions
   3.6.3 An Argument for the Rejection of Actor Notation
   3.6.4 Limitations of Functions
   3.6.5 Ranges and Comparisons in C-GRASS

3.7 Conceptual Graph Issues IV: Constraints in C-GRASS
   3.7.1 Canonical Constraints in C-GRASS
   3.7.2 Other Constraints

3.8 Summary

4.1 Chapter Objectives

4.2 Overall System Operation

4.2.1 Evolution of Storage System
4.2.1.1 Move From Binary Relational to N-ary Relational Strategy
4.2.1.2 Development of Domain Mappings ($\mu_R$)
4.2.1.3 Development of Adaptive Strategy
4.2.1.4 Metatyping and the Storage of Generic Referents
4.2.1.5 Some Intractable Problems Remaining

4.2.2 Parsing of User Input
4.2.2.1 A BNF Grammar

4.2.3 Development of Main-Memory Structures
4.2.3.1 Description of Graph-Building Structures
4.2.3.2 Example of Graph-Building Structures
4.2.3.3 Description of Lattice-Building Structures
4.2.3.4 Example of Lattice-Building Structures
4.2.3.5 Description of Definition-Building Structures
4.2.3.6 Structures in Early Versions of C-GRASS

4.2.4 Development of Basic Operations
4.2.4.1 Lattice Searching
4.2.4.2 Processing of Canonical Information: ‘Templates’
4.2.4.3 Projection
4.2.4.4 Development of Maximal Join
4.2.4.5 Expansion and Contraction

4.3 ‘Desirable But Incomplete’: Some Loose Ends
4.3.1 Parallelism and C-GRASS
4.3.2 Proof and C-GRASS

4.4 Summary: Why Implementation is Vital

5. Evaluation of C-GRASS

5.1 Chapter Objectives

5.2 Identification of Key Functions in C-GRASS
5.2.1 Main-Memory Functions Under Test

5.2.2 Interface Functions Under Test

5.3 Testing of Key Functions

5.3.1 A Basic Unit of Testing

5.3.2 Lattice Searching
  5.3.2.1 Description of Testing
  5.3.2.2 Predicted Results
  5.3.2.3 Results and Analysis

5.3.3 Maximal Expansion
  5.3.3.1 Description of Testing
  5.3.3.2 Predicted Results
  5.3.3.3 Results and Analysis

5.3.4 Projection
  5.3.4.1 Description of Testing
  5.3.4.2 Predicted Results
  5.3.4.3 Results and Analysis

5.3.5 Maximal Join
  5.3.5.1 Description of Testing
  5.3.5.2 Predicted Results
  5.3.5.3 Results and Analysis

5.3.6 Copy
  5.3.6.1 Description of Testing
  5.3.6.2 Predicted Results
  5.3.6.3 Results and Analysis

5.3.7 $\phi_0$
  5.3.7.1 Description of Testing
  5.3.7.2 Predicted Results
  5.3.7.3 Results and Analysis

5.3.8 Posting
  5.3.8.1 Description of Testing
  5.3.8.2 First Posting Test
  5.3.8.3 Second Posting Test
5.4 Correctness and Efficiency of Overall System 253
  5.4.1 Lattice Construction 253
  5.4.2 Canonical Model Construction 254
  5.4.3 Definition Construction 255
  5.4.4 Application Graph Construction 255
  5.4.5 Evaluation of Inquiry 257
5.5 How Variations in Application Domain May Affect Efficiency 258
  5.5.1 Lattice Complexity 258
  5.5.2 Granularity of Underlying Relational Storage 258
  5.5.3 Cardinality of Underlying Relational Storage Schemes 259
  5.5.4 Complexity of Application Graphs 259
5.6 Commentary on Evaluation 260

6. Conclusions and Future Work 261
6.1 Summary of Achievements 261
  6.1.1 Primary Aim: Conceptual Graphs as a Conceptual Schema Definition Language 261
  6.1.2 Aim #2: Correspondence Between Conceptual Graphs and the Relational Model 261
  6.1.3 Aim #3: Use of Extended Conceptual Graphs in a Hybrid System 262
  6.1.4 Aim #4: Increased Declarativeness of a Hybrid System 263
  6.1.5 Aim #5: Use of Existing Technology in Hybrid System Construction 263
  6.1.6 Aim #6: Investigation of Legacy Database Reuse in a Hybrid System 264
6.2 Significance of This Research 264
  6.2.1 Extended Conceptual Graphs Formally Map to the Relational Model 264
  6.2.2 Practical Implementation of Extended Conceptual Graph Theory 265
  6.2.3 Very Declarative Nature of Implementation 265
6.3 Unresolved Issues and Future Directions 265
  6.3.1 Some Unresolved Issues in Extended Conceptual Graph Theory and C-GRASS 266
    6.3.1.1 Reasoning Mechanisms 266
    6.3.1.2 Limitations of Constraints 266
    6.3.1.3 Algorithmic Comparison with Hypergraphs 266
  6.3.2 Future Directions 267
    6.3.2.1 Additional Database Capabilities 267
Appendices

Appendix A  Catalogue of Posting Events
Appendix B  C-GRASS User Manual
Appendix C  Notes on Parallel Extensions to C-GRASS
Appendix D  [Bowen and Kocura 1993]: Paper presented to the First International Conference on Conceptual Structures, Quebec City
Appendix E  [Bowen and Kocura 1994]: Paper withdrawn from the Second International Conference on Conceptual Structures (although a summary was presented)
Appendix F  Evaluation Test Programs
Appendix F  Building a Domain: Familial Relationships

List of Figures

1.1  A Loose-Coupled Hybrid Architecture 27
1.2  A Tight-Coupled Hybrid Architecture 29

2.1  Showing Domain Mappings in the Display Form 44
2.2  Domain Mappings in a Simple Conceptual Graph 45
2.3  Hypergraph of Figure 2.2 46
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.17</td>
<td>Multiple Non-Aristotelian Inheritance</td>
<td>138</td>
</tr>
<tr>
<td>3.18</td>
<td>Legal Referent Constructions in [Sowa 1993]</td>
<td>141</td>
</tr>
<tr>
<td>3.19</td>
<td>A Domain-Extended Conceptual Graph and $\phi_\Delta$</td>
<td>152</td>
</tr>
<tr>
<td>3.20</td>
<td>Subgraphs of Figure 3.19</td>
<td>154</td>
</tr>
<tr>
<td>3.21</td>
<td>Lists of $\phi_\Delta$</td>
<td>155</td>
</tr>
<tr>
<td>3.22</td>
<td>View Created From Figure 3.19 by $\phi_\Delta$</td>
<td>156</td>
</tr>
<tr>
<td>3.23</td>
<td>An Asserted Graph and How Memberships Affect It</td>
<td>160</td>
</tr>
<tr>
<td>3.24</td>
<td>A Graph Extended by Cardinality and Membership</td>
<td>160</td>
</tr>
<tr>
<td>3.25</td>
<td>An Illustration of Structurality</td>
<td>162</td>
</tr>
<tr>
<td>3.26</td>
<td>Adding Information to the Canonical Model</td>
<td>164</td>
</tr>
<tr>
<td>3.27</td>
<td>Graph Disconnection in the Canonical Model</td>
<td>165</td>
</tr>
<tr>
<td>3.28</td>
<td>Checking for Subtyping in the Canonical Model</td>
<td>166</td>
</tr>
<tr>
<td>3.29</td>
<td>Four Sorts of Conceptual Graph</td>
<td>172</td>
</tr>
<tr>
<td>3.30</td>
<td>Actors and Computation</td>
<td>175</td>
</tr>
<tr>
<td>3.31</td>
<td>An Addition Function</td>
<td>176</td>
</tr>
<tr>
<td>4.1</td>
<td>Early Binary Relational Structures of C-GRASS</td>
<td>185</td>
</tr>
<tr>
<td>4.2</td>
<td>Repository Schemes of C-GRASS</td>
<td>186</td>
</tr>
<tr>
<td>4.3</td>
<td>A Simple Conceptual Graph</td>
<td>198</td>
</tr>
<tr>
<td>4.4</td>
<td>Internal Representation of Figure 4.3 in C-GRASS</td>
<td>199</td>
</tr>
<tr>
<td>4.5</td>
<td>A Lattice and its Internal Representation in C-GRASS</td>
<td>201</td>
</tr>
<tr>
<td>4.6</td>
<td>Early Canonical Template 'Metagraphs'</td>
<td>203</td>
</tr>
<tr>
<td>4.7</td>
<td>Metagraph Properties as Application Graphs</td>
<td>204</td>
</tr>
<tr>
<td>5.1</td>
<td>A Basic Five-Dyad Unit and Some Graphs Constructed From It</td>
<td>215</td>
</tr>
<tr>
<td>5.2</td>
<td>Predicted Results for Lattice Searching</td>
<td>218</td>
</tr>
<tr>
<td>5.3</td>
<td>Test Results for Lattice Searching</td>
<td>219</td>
</tr>
<tr>
<td>5.4</td>
<td>Predicted Results for Maximal Expansion</td>
<td>222</td>
</tr>
<tr>
<td>5.5</td>
<td>Test Results for Maximal Expansion</td>
<td>223</td>
</tr>
<tr>
<td>5.6</td>
<td>Predicted Results for Projection Operation</td>
<td>226</td>
</tr>
<tr>
<td>5.7</td>
<td>Test Results for Projection Operation</td>
<td>228</td>
</tr>
<tr>
<td>5.8</td>
<td>Correct Result for $u = 10$, $v = 10$ and $j = 5$ (Maximal Join)</td>
<td>230</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.9</td>
<td>Predicted Figures for Maximal Join Operation</td>
<td>232</td>
</tr>
<tr>
<td>5.10</td>
<td>Graphical Interpretation of Figure 5.9</td>
<td>233</td>
</tr>
<tr>
<td>5.11</td>
<td>Test Figures for Maximal Join Operation</td>
<td>234</td>
</tr>
<tr>
<td>5.12</td>
<td>Graphical Interpretation of Figure 5.11</td>
<td>235</td>
</tr>
<tr>
<td>5.13</td>
<td>Predicted Results for Graph Copying</td>
<td>237</td>
</tr>
<tr>
<td>5.14</td>
<td>Test Results for Graph Copying</td>
<td>238</td>
</tr>
<tr>
<td>5.15</td>
<td>A Sample $\phi_A$ Test Prediction</td>
<td>240</td>
</tr>
<tr>
<td>5.16</td>
<td>Predicted Results for $\phi_A$</td>
<td>242</td>
</tr>
<tr>
<td>5.17</td>
<td>SQL Instruction of Figure 5.15</td>
<td>243</td>
</tr>
<tr>
<td>5.18</td>
<td>Test Results for $\phi_A$</td>
<td>244</td>
</tr>
<tr>
<td>5.19</td>
<td>Test Domain Before Execution of Posting Test #1</td>
<td>246</td>
</tr>
<tr>
<td>5.20</td>
<td>SQL Instructions Produced by Posting Test #1</td>
<td>247</td>
</tr>
<tr>
<td>5.21</td>
<td>Test Domain After Execution of Posting Test #1</td>
<td>248</td>
</tr>
<tr>
<td>5.22</td>
<td>Test Domain Before Execution of Posting Test #2</td>
<td>250</td>
</tr>
<tr>
<td>5.23</td>
<td>SQL Instructions Produced by Posting Test #2</td>
<td>251</td>
</tr>
<tr>
<td>5.24</td>
<td>Test Domain After Execution of Posting Test #2</td>
<td>252</td>
</tr>
</tbody>
</table>
CHAPTER 1
Introduction

The primary objective of a data model is the representation of an application domain - or a "slice of reality" as [Vassiliou 1986] dubs it. The problem is that no data model can ever really represent any "slice of reality" perfectly - a (perhaps) apocryphal story concerning Lewis Carroll relates that he liked to tell people about his 'perfect' map of England - perfect, because it showed every city, every town, every street, every house, every tree, every leaf on every tree, every molecule in every leaf, and so on. The map was perfect because it was on a one-to-one scale with what it mapped - England - but it wasn't particularly useful as a map because a map was needed to understand it.

Given that perfect representation of a domain is probably impossible, the success and applicability of a data model can be more usefully related to the ease with which the user may model and manipulate a given application domain. By this yardstick, the relational model [Codd 1970] has been the most successful model to date, achieving a wide acceptance among database users. Although the earliest implementations of this model were not particularly efficient, it has grown and adapted to the point where "it is generally the model of choice for the implementation of new databases" [Ullman 1988].

The strictures of the relational model make it particularly good for the representation and efficient processing of the data typically found in business applications - large quantities of fixed-format data with little structure or semantics. Codd himself has stated that the model was deliberately designed to be coherent yet simple, so that its concepts would be easily communicable as well as mathematically sound.
1.1 Advantages of the Relational Model Over Earlier Models

It seems to be true that there is no 'best' data model - only data models that are 'most appropriate for a task'. The failure to recognise this fact has been a particularly fruitless bone of contention between the relational model and rival approaches such as the object-oriented data model - even though the conceptual differences between the two are not particularly great [Stonebraker 1991]. If there is no 'best' model, it follows that some domains are better suited to representation in the relational model, whilst others are better suited to other models.

For example, take the storage of engineering design data in the relational model - such domains have large numbers of data descriptors, many scalar attributes and millions of parts. There might also be many different versions of the database to be maintained, and the data in such domains is typically complex. The relational model is not suited to this sort of domain, whereas models such as those based on the object-oriented paradigm might be. Conversely, the modelling of banking or accountancy packages by an object-oriented approach would probably be less successful than a relational approach, considering the high homogeneity of the domain.

Even though there may be no 'best' model, the relational model has swept all other models before it. This may be due to the importance of business applications, but is also due to the fact that the relational model is more coherent than the rival models extant at the time of its development. It was a major improvement upon other data models because it was not ad hoc in its theory - conversely, the network and hierarchical systems were developed in response to certain problems, but had no theoretical basis. Codd showed that the mathematical theory of relations could be applied to the various relationships in an abstract data structure. He was therefore able to bring a proven mathematical correctness and rigour to database structure.

1.1.1 Importance of Logical Data Organisation

The relational model differs from other models in that its primary concern is with logical data organisation - this is in contrast with earlier models, which failed to separate the question of physical storage from that of logical storage, and so required the user to possess considerable
knowledge of physical data organisation. One of the primary objectives of the relational model was to increase data independence - in other words, to protect application programs from physical changes to file organisation, which would in turn alter access paths to stored data. In order to achieve this objective, the user is shielded from physical storage matters, and so only needs to have knowledge of the logical data organisation.

1.1.2 Conceptual Simplicity

The relational model is conceptually simpler than other models - both in terms of the structures needed to represent a domain and the operations needed to usefully manipulate them. The only essential structure of the relational model is the relation scheme and because of this, the model only needs a small set of operations - the relational algebra - with which to manipulate schemes. This is in contrast to models that need multiple structures - such as records and pointers - where a separate set of operations is required for each data structure usable by the system.

Relational algebra is a formal method of manipulating schemes in the same way that arithmetic is a formal method of manipulating numbers. Both of these systems are closed - application of arithmetic operations to numbers always produces other numbers; likewise, the application of the relational algebra to relation schemes always produces other relation schemes. This closure property also allows composition - because the only structures present are relation schemes, the output of one operation can be used as the input of another operation; there is no need for predetermined access paths, which prevent operational composition in pre-relational models.

1.1.3 Declarativeness

The shift of organisational emphasis from partially physical to exclusively logical means that the relational model is far less navigational than other models. By this, it is meant that the user only needs to know where the data was in the logical representation of the domain, not where it

1 This is most definitely the case with emerging object-oriented database systems, which use completely general-purpose programming languages and require each data type to have its own set of definition and manipulation routines.
is physically stored. In addition, the user also needs to know how to connect relation schemes together if data is dispersed over a number of such schemes.

The relational model is therefore far more *declarative* in how it manipulates data, allowing the user to specify *what* is required rather than *how* to get it. This important difference between the relational model and its rivals is memorably described by [Titman 1974]: “when Christopher Columbus crossed the Atlantic, he had to do a lot of navigation. When I cross the Atlantic, I sit in an armchair in a 747, and I know which I prefer”.

1.2 Disadvantages of the Relational Model

As [Date 1990] has observed, “the relational model is *not* a panacea”. It has a number of weaknesses and omissions - sometimes intentional, sometimes not - that effectively limit the set of application domains that it can process efficiently. It can be argued that the relational model is a victim of its own success - as its popularity and reliability have increased, the number of application domains that people want to represent in the relational model has also increased. The weaknesses of the model have been brought into sharp relief by this trend, to the point where there is a distinct yet increasing class of application domains that are difficult or near-impossible for the model to represent and manipulate.

These domains can be characterised by (i) the need for fast retrieval of data, (ii) more complex structures and data types than the relational model, and (iii) the need for more powerful functionality than the relational model can offer.

1.2.1 Difficulties of Modelling Complex Structures

The only structure required by the relational model is the relation scheme. Within such schemes, implementations of the relational model generally tend to support only a small set of predefined attribute types - such as integer, character string, Boolean, and so on. This may be acceptable for business-style application domains that require only simple data descriptors to handle large
numbers of tuples, but many new application domains (such as CAD and other engineering systems) require the ability to represent complex structures and to be able to treat them as discrete units when processing; they also tend to require a much larger set of data descriptors than the relational model can offer. [Smith and Smith 1977] seem to have been the first to highlight the fact that relational storage is ill-suited to the modelling of certain domains, such as those referred to above.

Because the relational model may fragment data between relation schemes in order to achieve a normalised database, this can be time-costly when trying to work with them. In the words of [Mylopoulos and Brodie 1990], "it is just plain hard to represent complex objects with tuples ... and just plain costly to disassemble and reassemble complex descriptors". Although a complex domain is hard to represent in any model, the simplicity of the relational model does give rise to increasingly severe performance overheads as structural complexity increases; complex data types can be built up in the relational model, but this construction must be done in a linear fashion [Stone and Hentchel 1990] and the operations upon these new data types are still limited to the relational algebra. These factors have sparked variant relational models such as non-first normal form, which can model nested structures, and extensible systems that support abstract data types.

1.2.2 Limitations of Relational Manipulation Languages

Expressive power tends to be limited by the limitations of the storage structures used by a given model; for example, the hierarchical model was geared to the efficient storage and manipulation of hierarchical information. Because relational structures experience difficulty when modelling certain complex domains, it can be concluded that relational manipulation languages also experience difficulty when manipulating those same domains.

1.2.2.1 Logical Basis of Limitations

Those domains that are badly served by the relational model require more powerful operations than the relational model can supply. As their structures are extended, they require a
commensurate extension of manipulative power in order to process them. The relational model, in order to maximise efficiency of operation, restricts itself to a small set of operations that can be efficiently optimised. Although this restriction has a sound logical basis ([Gallaire et al 1984] contains a useful treatment of this issue), its result is that the capability of relational manipulation languages to perform non-algebraic computation upon its own structures, let alone more complex ones, is limited. This is because they are not capable of performing computations that a general-purpose programming language is capable of.

In an attempt to increase its computational power and so be able to model more domains, most implementations of the relational model can embed themselves in general-purpose programming languages. The problem is that general-purpose programming languages aren’t very good at handling sets of tuples, and so special ‘cursors’ have to be invoked in order to take the tuples into the programming environment, one at a time. This difference between set-based and programming environments is the famous impedance mismatch [Ullman 1988].

1.2.2.2 Lack of Transitive Closure of a Binary Relation

Finding all the managers of Jane, whether they be direct or indirect, is beyond the capabilities of the relational algebra. The trouble with such a query is that it is open-ended, and there is really no way of knowing when the answer to the query is complete.

Such queries cannot be expressed in the relational algebra because they cannot be formulated as definite, optimisable statements in that language. This is due to the absence of the least fixpoint operator from the original relational algebra, and many authorities consider this operator to be an essential addition to relational manipulation languages. This may explain the rather dubious rise of Prolog and its subsets (which can perform transitive closure calculations fairly easily) as a database manipulation language.
1.2.2.3 **General Lack of Deductive Retrieval**

As the relational model is completely extensional in nature - and because it possesses no logic outside that upon which the relational algebra is based - it possesses only the crudest facility for the derivation of intensional information. Relational views can be seen as an intensional mechanism - for example, [Date 1990] gives a view for the intensional derivation of Grandmother-Of:

```sql
CREATE VIEW GRANDMOTHEROF (GRANDMOTHER, GRANDDAUGHTER) AS SELECT M1.MOTHER, M2.DAUGHTER
FROM MOTHEROF M1, MOTHEROF M2
WHERE M1.DAUGHTER = M2.MOTHER;
```

from the Mother-Of scheme:

```sql
MOTHEROF (MOTHER, DAUGHTER)
```

Unfortunately, this crude mechanism is not really sufficient for more than the simplest deductions. Considering that the intensional derivation of hierarchies is useful in many application domains ('is-a' hierarchies govern inheritance in many artificial intelligence applications as well as business applications), this is a serious deficiency.

1.2.2.4 **Lack of Reasoning with Metadata**

The query ‘does a supplier x exist?’ is akin to asking ‘is there a relation in which there is a tuple that tells us if the supplier x exists?’. This requires reasoning about relation schemes themselves, not the tuples in them. This second-order reasoning is outside the scope of the relational model’s manipulative capabilities.
1.2.2.5 Lack of Formal Computational Basis

Although many implementations of the relational model support both row-wise (arithmetic) operations and column-wise (aggregate) operations, these are strictly outside the range of pure relational algebra and so have no formal basis.

1.2.3 Semantic Limitations of the Relational Model

Any “slice of reality” has semantic constraints that prevent its extension by data that would cause it to become inconsistent. Data models have similar sorts of constraint mechanism, but the semantic constraints of the relational model are notoriously minimal.

1.2.3.1 Representation of Negative Information

A drawback of the relational model is that it is completely extensional in nature, being based upon the Closed World Hypothesis [Gallaire et al 1984]. The assumption is made that if a tuple is not stored, then it is false. Although some domains are naturally closed worlds (such as airline booking systems, where nothing outside the database can be true, and non-existent reservations really are false), many domains are open world (in which if something is not explicitly declared as true or false, then it is unknown).

The presence of intensional information means that a database doesn’t necessarily have to store a complete extension. All that has to be stored is a collection of rules and a subset of the extension from which the complete extension can be derived by rule application. Many domains contain unknown information, but the relational model cannot really support such information and so does not model those domains with much success.

1.2.3.2 Representation of Disjunctive and Indefinite Data

There is no way to represent information with an uncertain truth value - for example, the relational model wouldn’t be able to handle the fact ‘either AnyCo or AnyLtd supplies the
Widget part, but which one is unknown'. Similarly, if some attribute in a relation scheme is unknown, it is *null*. The AnyCo or AnyLtd example above is related to this: the fact that ‘Widget’ is supplied is known, but the supplier is currently uncertain. The status of such nulls in algebraic operations such as joins is uncertain, and the relational algebra has no hard and fixed approach to this problem at present.

1.2.3.3 *Representation of Hierarchical Information and Encapsulation*

The relational model has no implicit mechanism for the hierarchical organisation of its structures. For example (adapted from [Gray 1985]), consider these relation schemes:

```
PERSON   (P-ID PERSON-NAME ADDRESS)
STUDENT   (P-ID INSTITUTION YEAR)
TEACHER   (P-ID INSTITUTION TELEPHONE OFFICE)
LECTURER  (P-ID POSITION YEAR-OF-APPOINTMENT)
```

Every lecturer is also a teacher, so all tuples of the LECTURER scheme have a tuple in the TEACHER scheme and *inherit* properties from the tuple with the same key. Similarly, all students and all teachers are people and so all tuples of the TEACHER and STUDENT schemes have a tuple in the PERSON scheme - again inheriting from the tuple with the same key; if a person does not exist, then that person cannot be a student or a teacher; and if a teacher doesn’t exist, then that teacher can’t be a lecturer. These schemes are patently hierarchical in nature, yet the relational model has no proper way of handling this.

1.2.3.4 *Problem of Domain Join Mismatch*

Consider this example from [Beynon-Davies 1991], where two relation schemes are present - one is a shipping scheme and the other a scheme concerned with staff of the shipping line:

```
SHIP       (ID NAME DISPLACEMENT HOME-PORT)
EMPLOYEE   (ID NAME DEPARTMENT BIRTHDATE)
```
These two relation schemes might hold the following tuples:

**SHIP**
(1001 Victoria 50000 London)

**EMPLOYEE**
(6000 Victoria Personnel 27-07-1966)

Programmer error may join the two Name columns together, informing us that the director of Personnel displaces 50,000 tonnes. Although a programmer would have to be inept for this to happen, this does illustrate the point that there is no semantic limitation to prevent this in the relational model. [Codd 1990] has argued for the implementation of such a mechanism, but it has yet to appear in any commercial implementation.

1.2.3.5 Program-Level Semantics are Vulnerable to Abuse or Error

Because the level of semantic capture in the relational model is so low, any useful semantics tend to be specified by the system designer in the application programs associated with the database. This is because constraints are not part of the data and so are not stored with the data.

This is not a problem that is unique to the relational model, but plagues any model where data and meaning are separate [Stonebraker 1989]. Constraints tend to get buried deep in application programs where they are difficult to change, and the lack of a central 'semantic repository' in the model means that constraints may well be spread over several such programs; if those programs disagree on their semantics, inconsistencies can easily arise. [Sowa 1984] cites the case of a database system that was unable to answer queries about Puerto Rico consistently - one program handled it as a country, whilst another handled it as a state of the United States.

1.2.3.6 Summary of Semantic Capture in the Relational Model

The only semantic information available in an application domain represented by the relational model are attribute names and relation scheme names. There is no requirement for these names
to be meaningful (in which case the semantics of the implementation drop to just about zero),
and no constraint upon how the attributes in a database may join. This is obviously not
sufficient to meet the need for the handling of complex domains. The data dictionary shows
only how data is organised, and does not really contain much of semantic significance.

1.3 Extending the Relational Model

The relational model obviously has difficulty with certain sorts of domain, but can it be
extended in order that it might more easily model those domains?

1.3.1 Some Obvious Extensions

Obvious extensions would include greater expressive power, but the ability of the relational
model to embed itself in a host language isn’t really sufficient. It would be preferable to
implement extensions to relational languages that achieve greater functionality, whilst
preserving those things (optimisation, etc) that have made the relational model successful. More
expressive power would almost certainly need more complex structures over which they can
range; and tighter constraints on data would be necessary in order to maintain the consistency of
an application domain.

1.3.2 Some Less Obvious Extensions

Apart from these ‘obvious’ extensions, there are a number of more subtle properties that,
although concerned mostly with the user’s perception of the model, would also be important.

1.3.2.1 Declarativeness Again

The first of these is related to the issue of declarativeness. In comparison to earlier data models,
the relational model has a declarative manipulation language that allows the user merely to
“...specify only what data is wanted, not a procedure for obtaining that data” [Date 1982].

11
Therefore, it follows that the more declarative a manipulation language is, the less need there is for application-specific navigational information - that is, there is less need for the user to know which structures contain which information, what they are called, and what to do in order to get the information required out of those structures.

However, the relational model still needs some implementation-specific knowledge in order to function over normalised schemes; for example, consider the following database scheme of an investment house from [Bowen and Kocura 1994] (reprinted as Appendix E in this thesis):

\[
\begin{align*}
OB & \quad (OFFICE\ BROKER#) \\
IB & \quad (INVESTOR#\ BROKER#) \\
SD & \quad (STOCK\ DIVIDEND) \\
SIQ & \quad (STOCK\ INVESTOR#\ QUANTITY)
\end{align*}
\]

and the query: "which investors do business with Broker #100, what stocks do they have, and how many of each?". In SQL, the query would be fairly simple:

\[
\begin{align*}
\text{SELECT} & \quad (\text{FIRST.INVESTOR#}, \text{FIRST.STOCK}, \text{FIRST.QUANTITY}) \\
\text{FROM} & \quad \text{SIQ FIRST, IB SECOND} \\
\text{WHERE} & \quad \text{SECOND.BROKER#} = 100 \\
& \quad \text{AND} \quad \text{SECOND.INVESTOR#} = \text{FIRST.INVESTOR#};
\end{align*}
\]

Three problems with this query can be noted. Firstly, the location of the data must be specified, with the user needing to know the logical location of the data domains involved in the query. Secondly, relation and domain names must be known, with the user needing to know the exact names of the data domains and relation schemes involved in the query. For example, OFFICE and DIVIDEND are recognised domain names, and SIQ is a recognised relation scheme name. Thirdly, the syntax is very rigid - SQL and QUEL have a rigid syntax that a query must adhere to; they don't allow any margin for error. Without this knowledge, novice users are lucky if they can formulate even the simplest of queries and are constrained by a lack of knowledge about the internal representation of data.
Although the relational model has perhaps the most declarative language commonly available, it can be argued that it is still not declarative enough. All other things being equal, users seem to prefer declarative languages over imperative languages and so it is sensible to conclude that they would prefer a more powerful (yet still declarative) data manipulation language rather than a very powerful (yet still procedural) data manipulation language. Any extensions to the relational model should therefore at least preserve its declarative property and increase it if possible.

1.3.2.2 High-Level Representation

Any extended system with increased semantic capture should have the ability to specify those semantics in as abstract a way as possible, in order to make the modelling of that information as close to the application domain as possible. The same property also applies to the specification of constraints. As such an abstract system should be as close to the application domain as possible, the amount of implementation-specific knowledge required by the user should be minimal.

1.3.3 Appearance of Postrelational Systems

Systems which go some way towards meeting the aims specified have begun to appear, and there has been "a renaissance of research activity building 'next generation prototypes' which attempt to rectify the drawbacks of current relational systems" [Stonebraker 1989]. These prototype systems are based on a number of new data models possessing structures and operations that are capable of modelling complex domains more naturally. These new postrelational models can be seen as part of a continuing trend away from expressly physical considerations towards logical considerations. Some postrelational models attempt to extend the relational model in ways that will allow the above aims to be met. Such systems are referred to here as extended relational models. Other approaches have no allegiance to the relational model.

Postrelational data models can also be usefully categorised into two groups: those that offer highly abstract methods of achieving greater semantic capture than the relational model
(semantic data models) and those models that, in addition to an increased level of semantic capture, also offer markedly greater operational functionality than the relational model (object-oriented data models). As these two areas tend to overlap to some degree - semantic data models are object-oriented in the sense that they manipulate objects rather than records, and object-oriented data models do have increased semantic capture, semantic data models are used in this context to refer to those models that are primarily concerned with representational issues, whilst object-oriented data models are used to refer to those models that are primarily concerned with procedural and behavioural issues.

1.3.4 Object-Oriented Data Models

The term 'object-oriented data model' is perhaps something of a misnomer, as object-orientation is more of a design philosophy than an actual data model. As an idea, object-orientation can probably be traced back to Simula-67; this programming language contained rudimentary forms of what are now considered to be two of the core features of object-orientation - object identity and communication between objects. Later languages such as SmallTalk developed these ideas further and added features such as property inheritance and class definition. Current object-oriented languages - such as C++ - have all of these features.

Early object-oriented database systems were nothing more than advanced file managers, but these have developed into proper database managers that can share data across applications. Examples of commercially available systems based on this approach are O₂ [Deux et al 1990] (which uses SmallTalk), ODE [Agrawal and Gehani 1989] (which uses C++), and the extended relational POSTGRES [Stonebraker et al 1990]. [Ahmed et al 1992] offers a useful comparative study of a number of these systems.

1.3.4.1 Basic Object-Oriented Philosophy

The basic idea behind object-oriented systems is that the user should be able to use structures and operations that closely match their domain counterparts, rather than having to work in terms of artificial constructs such as relations and other unnatural record types; as [Date 1990] has
noted, "the fundamental idea is to raise the level of abstraction". In order to achieve this, object-oriented data models view the application domain as a series of data types or classes - each of which has:

- a series of attributes, which may themselves be objects.
- a series of access and other manipulation operations (or methods) over those attributes; each class therefore has its own set of operators and embedded semantics, which centralises the semantics of the domain (making maintenance easier); and as programs to not directly access data attributes but use method access instead, this tends to reduce the amount of additional code needed.

Classes are rather hierarchical in nature and can inherit attributes and methods from others. An object-oriented database is a series of class instantiations or objects - much as a relational database is a series of scheme instantiations. It is manipulated by the application of methods to objects, and this is achieved by the issuing of messages to objects by the user (initially), or from one object to another; the query language of such systems is completely general-purpose.

1.3.4.2 Advantages of Object-Orientation

The undoubted advantage of systems based on the object-oriented philosophy over other data models is their computational completeness - as the manipulation languages of such systems are general-purpose, any structure and any manipulation can be represented and limitations are effectively set by the ingenuity of the user. Object-oriented data models also use just one language to do everything - unlike relational databases, which need to invoke a general-purpose host language in order to perform more complex manipulations. Object-oriented databases do not therefore suffer from the classical relational problem of impedance mismatch.

1.3.4.3 Disadvantages of Object-Orientation

Raw computational power without impedance mismatch may seem to be a considerable point in favour of object-oriented database systems, but is it really so useful? Although computationally
complete, object-oriented languages are, when all is said and done, programming languages and are therefore completely imperative. This is intolerable to the casual user, as a heavy investment in both object-oriented design principles and knowledge of a suitable manipulation language is required. This lack of a declarative, non-procedural language is such a serious problem that many object-oriented systems offer SQL-like languages as syntactic sugar for their imperative languages - examples of such languages include the CQL++ interface to ODE [Agrawal and Gehani 1989] and the CO₂ interface to O₂ [Deux et al 1990].

Although the problem of impedance mismatch is solved by object-oriented systems, the solution of using a host language is not particularly attractive, as it removes the strengths of the relational model - efficient, optimisable performance over large datasets. It is perhaps more preferable to extend the set-based operations of the relational model, rather than extending the record-level, tuple-at-a-time host language.

Object-oriented systems also require the user to perform the vast majority of structural and manipulative implementation. This almost certainly means that there is no scope for optimisation, at least as it is understood in relational circles. The semantic capture of object-oriented systems - primarily hierarchical inheritance and class behaviour - is also not much better than that of the relational model, and a high price is paid in other areas for this - somebody still has to write the methods to define the semantics of a given object, and this may still not be abstract enough for the casual user². Finally, any operations created may not be closed; thus, composition of operations in an ad hoc query may be difficult.

These are just a selection of many objections that have been raised with respect to object-oriented data models. Many more problems are currently “swept under the carpet” [Date 1990].

1.3.4.4 Fuzziness of Object-Orientation

Another worrying property of the various object-oriented data models is that, although there are a number of object-oriented systems currently on the market, an object-oriented data model

² To be fair, it must be said that relational systems are equally bad at this.
doesn’t formally exist. There has been no hard and fast definition of what the object-oriented data model should contain for a long time, and it has been noted by [Stonebraker 1989] that the six expert participants in a panel discussion on object-oriented databases (at the 1987 Conference on Very Large Data Bases) managed to disagree completely on exactly what an object-oriented database might be; the object-oriented properties described above may be considered as ‘core’ properties that more or less everybody agrees with.

This lack of a formal model has led many to compare object-oriented data models with the earlier hierarchical and network models\(^3\). Given that they possess complex structures and need a navigational query style to move around them, this is probably not surprising. However, the extended relational NFN\(^3\) (non-first normal form) model [Roth et al 1988] has a similar structural aim to what are generally regarded as object-oriented systems, whilst still retaining a formal model that is an extension of the relational model to handle nonatomic tuples and nonatomic attributes (by which it is meant that each tuple can have an attribute that is a set of those attributes, whilst each attribute of a relation scheme can be composed of multiple subattributes\(^4\)). Given this object-oriented extension to the formal basis of the relational model, it is not surprising that the lack of formalisation in more regular object-oriented models is slowly being rectified in key areas [Bertino et al 1992] by using relational-style languages with a mathematical basis (an object-oriented predicate calculus) that is based in large part on the relational calculus.

1.3.4.5 Common Ground Between the Relational Model and Object-Orientation

Relational and object-oriented systems seem to be moving towards some sort of common ground: [Kim 1991] has claimed that “solutions to most of the data modelling related difficulties of conventional database systems are inherent in an object-oriented data model”, and that object-oriented data models can be considered as a superset of the relational model. Given

\(^3\) Hierarchical systems tend to have preferred access paths, along which queries are fast, but any query off those access paths are notoriously inefficient. For example, consider a hierarchy of students and the classes they perform; it is very easy to find all the classes that a particular student takes as this is a preferred path, but it is substantially more difficult to find all the students who take a particular class as it is not on a preferred path [McHenry 1993].

\(^4\) Effectively, each ‘cell’ of a relation scheme can be another relation scheme.
the number of similarities between the relational and object-oriented approaches, it could be argued that the object-oriented paradigm is just a significant contributor to the relational model, rather than a separate model in its own right (see [Blaha et al 1988, 1994; Premerlani et al 1990] for some interesting mappings between the two approaches). Most relational vendors are already adding object-oriented extensions to their systems (it is far easier to extend the tried-and-tested relational model with object-oriented ideas than it is to extend object-oriented systems with relational storage), and many object-oriented systems (such as OpenODB [Ahad and Dedo 1992]) are based on relational storage. These 'object-relational' systems combine object-oriented ideas with the brute retrieval power of the relational model. SQL is also forcing convergence at the interface level between object-oriented and relational systems, and so the boundary between pure relational systems and pure object-oriented systems is becoming more blurred as time passes.

Is this common ground any sort of panacea for the problems of the relational model? Probably not, although these hybrid systems - about which more will be said when discussing logic and the relational model later on - offer a lot of power and are ideal for the expert user who has the requisite knowledge to use them properly.

1.3.5 Semantic Data Models

Unfortunately for object-oriented systems, not all users are expert users. From the casual user's point of view, the most serious drawback of an explicitly object-oriented approach is that its implementations are not declarative and not abstract in their semantic capture; instead, they are preoccupied with offering the user the ability to define any structure and any operation. Conversely, semantic data models (the first of which appears to be [Abrial 1974]) are concerned with semantic capture of as much of a domain as possible, and with the representation of that domain in a high-level and abstract form.

As [Hammer and McLeod 1981] have remarked, "a database whose organisation is based on naturally occurring structures will be easier for a database designer to construct and modify than one that forces him to translate the primitives of his problem domain into artificial constructs".
The conceptual simplicity of the relational model means that it does not have the wherewithal to handle anything more than simple structures or semantics — such as certain dependency relationships between simple atomic data. For example, knowing that two numeric values are different in kind — such as ship tonnage and a person's salary — and should not be joined is not possible at the representational level in the relational model.

1.3.4.1 Basic Semantic Data Model Philosophy

To paraphrase [Brodie and Mylopoulos 1986], any semantic data model must have a rich semantic theory for relating the information it stores to its subject matter. Analysis of the various approaches available ([Peckham and Maryanski 1988] give a very useful description and comparison of eight semantic data models, and [Hull and King 1987] give an excellent overview) suggests a number of concepts that are central to the task of semantic modelling:

- like the object-oriented model, the application is represented by objects, which are split into identifier and attributes - the identifier distinguishes objects from each other, whilst the attributes describe the properties of the object and its relationships to other objects.

- higher-order relationships than those offered in the object-oriented model are supported:

  • **generalisation** [Smith and Smith 1977]: the organisation of classes into hierarchies. For example, VEHICLE may be a generalisation of UNDERWATER-VEHICLE, which may itself be a generalisation of SUBMARINE. The higher-level object *encapsulates* the lower-level ones.

  • **aggregation** [Smith and Smith 1977]: as the relational model may take several attributes and aggregates them to form a relation scheme, semantic data models may take several objects and aggregate them together into more complex objects. For example, a SHIP object may be the aggregation of the ship's name, its captain (possibly another object in its own right, and so therefore also probably an aggregation), its tonnage, registry information, and so on.
• **classification** [Brodie 1986]: the generation of a general class from a number of objects. For example, all those VEHICLE objects that operate underwater can be classified as UNDERWATER-VEHICLE objects.

• **association** [Brodie 1986]: a membership property that groups all objects of a particular class (such as the set of all submarines, and so on).

### 1.3.4.2 Value of Increased Semantics

As the seminal paper by Codd [Codd 1979] has noted, "the task of capturing the meaning of data is a never-ending one ... the goal is nevertheless an extremely important one, because even small success can bring understanding and order into the field of database design". Any increase in the semantics that can be represented - such as those outlined above - should greatly increase the usefulness of a database system: "a meaning-oriented data model stored in a computer should enable it to respond to queries ... in a more intelligent manner".

### 1.3.4.3 Why Don't Semantic Data Models Rule the (Database) World?

Given the improved level of semantics, why are implementations of various semantic data models not as widespread as relational implementations? One valid answer [Peckham and Maryanski 1988] is that it takes time to develop implementations of a model - after all, the first research papers on the relational model took the best part of a decade to appear after the initial idea was mooted, and robust implementations took even longer. As the earliest references to semantic data models seem to date from the mid-1970s, it might be supposed that something usable might have appeared by now.

Perhaps a more plausible answer to this question is that semantic data models simply are not sufficient for the development of implementations. [Date 1990] identifies four steps in the construction of a semantic 'theory' (a la [Brodie and Mylopoulos 1986]):

i. identification of useful semantic concepts.

ii. derivation of formal objects.
Unfortunately, the vast majority of semantic data models have no manipulative component and so never pass the first or second stage of Date's sequence; only a very few (such as the functional data model [Buneman and Frankel 1979; Shipman 1981] and RM/T [Codd 1979]5) can truthfully be classified as being sufficiently developed for implementation. As [Codd 1979] has observed, recent work on semantic data modelling has "a strong emphasis on structural aspects, sometimes to the detriment of manipulative aspects. Structure without corresponding operators ... is rather like anatomy without physiology" (author's italics).

This deficiency means that most semantic data models are adequate for use in database design only - they have no internally consistent or formal way of manipulating their semantically rich structures themselves. The major impact of semantic models has therefore been in database design - the most prominent models being the entity-relationship model [Chen 1976] and related systems such as the extended entity-relationship model [Teory et al 1986] and NIAM [Leung and Nijssen 1987, 1988]. Chen's system appeared after the formal description of the relational model, and its diagramming system was quickly co-opted as a way of representing relational structures at a more conceptual level; the same is generally the case for the other systems mentioned.

1.4 Logic and the Relational Model

Although semantic data models do increase the semantic capture of an application domain, the actual increase that they achieve is not particularly impressive. Although they are movements in the right direction, their development has generally been rather ad hoc, without much formal

---

5 RM/T is a principled extension of the relational model in order to increase the relational model's semantics, with specific relation scheme types for entities and their properties, an improved set of integrity constraints, and a more complex system catalogue to keep track of such schemes. Unlike many other models, it possesses a fully realized manipulative component based on an extension of relational algebra.
development beyond representational structures, and this has led to the vast majority of those models being used as design systems.

1.4.1 A Model-Theoretical View

In sharp contrast, the development of the relational model is firmly grounded in first-order predicate logic and can be characterised as model-theoretical in nature [Reiter 1984]. From this point of view, any application domain modelled by a language - in this case, the relational calculus - admits at least one relational interpretation of that domain, in which all propositions are true. Such an interpretation is also referred to as a model (hence the term model-theoretical). This view of the relational model "has an elegance and simplicity that accounts, in large measure, for the overwhelming success of Codd's original proposal" [Reiter 1984].

1.4.2 A Proof-Theoretical View

[Reiter 1984] has shown that the model-theoretical viewpoint presented above can also be viewed in a proof-theoretical fashion by viewing the application domain not as a set of relations, but as a series of closed and well-formed formulas. In conjunction with additional axioms (specifically those of domain closure, uniqueness of name, equality and completion), those ground formulas form a first-order relational theory of the application domain that maps uniquely to an equivalent model-theoretical interpretation of the domain. The advantage of the proof-theoretical view over the more usual model-theoretical view is that the proof-theoretical view allows the use of the mechanisms of first-order predicate logic in order to model a domain.

Most of the problems of the model-theoretical approach are tractable from the proof-theoretical viewpoint although, as [Gallaire et al 1984] have pointed out, the proof-theoretical view does not directly make for particularly efficient systems - the combinatorial complexity of axioms such as completion and equality militate against such a direct implementation - but what it does do is point to the fact that an approach based upon provability of a theory is more amenable to extension than an approach based upon the truth of an interpretation.
1.4.3 Relevance of Proof-Theoretical View

[Reiter 1984] remarks that the proof-theoretical view of the relational model provides a natural way of tackling traditional problems of the relational model, and “provides a correct treatment of [extensions to] the relational model to incorporate more real world semantics”. A proof-theoretical view of an application domain can use a single language to represent data and constraints uniformly; as logic is the basis of all other models, it can be used to handle seemingly disparate problems such as dependencies, constraint enforcement, and so on. The most obvious way in which the relational model can be extended would therefore seem to be to increase its logical capabilities through a proof-theoretical approach; such a model would subsume the relational model and retain its sound logical basis and capacity for optimisation, whilst adding the capability to represent additional constraints on data that are not limited by the model-theoretical view.

This conclusion has given rise to a number of modelling approaches that are based on the first-order (and occasionally higher-order) predicate calculus. It is perhaps surprising that the movement towards the use of logic - as a means of defining the structures and constraints of application domains, as well as being a way of manipulating them - seems to date from as early as [Steele 1975], and predates approaches such as entity-relationship modelling.

1.4.4 'Hybridisation': Artificial Intelligence and Relational Databases

The logic-based systems that have appeared tend not to be based not on pure logic, but instead on the knowledge representations and techniques of artificial intelligence systems. Database systems generally tend to store many repetitions of similar data, and a small set of data types (the descriptors that make up the data dictionary) is sufficient to organise that data; apart from elementary formatting information, the data dictionary contains little or no semantics. In contrast, artificial intelligence systems operate over small sets of data, within which there are few repetitions; what passes for a data dictionary in such systems may conceivably contain a descriptor for each item of data, and so represents the opposite semantic extreme. As [Sowa 1984] has observed, the difference between artificial intelligence and database systems can be
measured by the ratio of descriptors to data. Most artificial intelligence systems don't distinguish much of their data from data descriptors, and so the ratio between the two is very low. In contrast, the number of data descriptors required by a database system is normally very small in comparison with the amounts of data stored.

However, neither of these independent approaches to data modelling has remained static, and the extreme difference between the ratios of the two systems has been narrowing. Relational databases have been shown to require greater semantic capture as they attempt to model ever more complex application domains, whilst artificial intelligence systems require more efficient algorithms for storage and retrieval as the size of the data sets they range over has gradually increased. If all the data of an artificial intelligence system is held in main memory, it may run out of space and so require some form of virtual memory handler (and also be extremely slow).

As the ratio between descriptors and data in both systems is becoming more similar, a trend towards some common ground can be identified in both approaches - towards a system that combines the high semantic capture of an artificial intelligence system with the mass storage and efficient manipulation of a database management system. This common ground therefore represents a midpoint between the two camps - where large sets of data have semantically useful descriptors and other constraints associated with them.

1.4.4.1 Potential Benefits of Hybridisation

The potential benefits of common ground are so important that [Brodie 1989] has predicted that "the effective application of Artificial Intelligence (AI) technology and the development of future computing systems requires the integration of AI and database technologies. The integration will benefit both parties and will substantially advance the state of computing ... Effective AI-database integration requires a deep understanding by AI people of what database technology could offer, and by database people of the requirements of AI systems".

Movement towards that common ground has led towards the development of what might be termed 'hybrid' or 'expert database' systems. Essentially, such systems employ both database
and artificial intelligence technology in the same system, using a database management component to handle the hard work of storage and retrieval and using an artificial intelligence component to impose semantics on that data [Manola and Brodie 1986]. The point of interest here is that the strengths of one component can be used to overcome substantial weaknesses of the other component - for example, consider this example of a submarine's definition:

- database definition: a submarine is a relation scheme with fields REGISTRY NUMBER, NAME, COMPLEMENT, NUMBER OF PROPELLERS, MAXIMUM SPEED, and so on;
- artificial intelligence definition: a submarine is a marine vehicle capable of operating underwater and usually equipped with torpedoes and a periscope.

The database definition of a submarine is that of a data dictionary - it has minimal semantic information and simply tells the system where certain fields appear and what their formats are - whereas the artificial intelligence definition is not tied to an implementation and simply tells the user what a submarine is. The first is useful if the efficient storage and retrieval of information about a lot of submarines is needed, and the second if reasoning about them is required (for example, the knowledge that every submarine is a marine vehicle allows us to rule out information about submarines in Trafalgar Square).

The advantages of combining such disparate approaches are obvious: the artificial intelligence component greatly increases the level of semantic information available to a database system - giving it a data dictionary that can handle issues such as abstraction, hierarchy, and such - whilst the database component allows retrieval of large sets of information.

1.4.4.2 Separation of Data and Semantics

A key article of any hybrid system is the separation of data and meaning: any item of information (as opposed to data) can be said to consist of both data and the meaning of that data, and the majority of hybrid systems separate the two - the data is pushed into secondary storage, whilst the semantics remains in main memory. If domains featuring large sets of data
are to be represented, then separation is generally advantageous because database retrieval techniques will operate much more efficiently over the large data set than other techniques operating over the same information where the separation has not occurred. In such systems, most of the querying is done at the database level because, as the underlying databases can be optimised, the performance of queries over large data sets degrades more gracefully than in systems where all information is stored in main memory. Because data is all that distinguishes many facts from one another, the core of 'skeletons' in main memory is minimal.

Because data has been separated from meaning, any query must reintegrate the two if useful processing is to occur. In most loose-coupled hybrid systems (see section 1.4.5 below), this requires communication between the artificial intelligence and database components, and this inevitably takes time; but if the application domain is without much data, then the overheads incurred by the necessity to compute calls and submit them to the database component is larger than if the information was not separated and was kept in main memory. Thus, a hybrid system is slower than a non-hybrid until a certain break-even point of data is reached, where the access times of the separated data and semantics are superior to the access times of the original information in main memory. Separation is also advantageous in terms of the speed of processing over pure semantics - it is generally quicker to perform pure reasoning over uninstantiated semantics than over instantiated semantics. In theory, hybrid systems should offer better brute retrieval power than pure artificial intelligence systems, as well as offering logical manipulation and constraint facilities that are considerably more advanced than those found in existing relational database management systems. Unfortunately, the interfacing of such components is not as simple as it might first appear, and this has led to a number of implementations that fail to utilise the database management component properly.

1.4.5 Approaches to Hybridisation

[Mylopoulos 1986] notes that hybridisation normally proceeds from either extreme of the continuum between database management systems and expert systems. Hybrid systems are

---

6 This term refers to facts without instantiation, such as 'a person has a taxcode'. This structure trades its data for mappings to places where all instances of persons with taxcodes are stored.
either database management systems that have been extended with various expert system features (for example, Postgres [Stonebraker 1987, 1990]), or are expert systems (such as KEE or LOOPS) that have been extended with various database management features. The efficiency of these approaches is related to the level of integration of the two components, which varies between implementations; both approaches are also evolutionary - they adapt existing technologies, but such approaches may be inadequate in the long term for anything other than research and development (although [Al-Zobaidie and Grimson 1987] state that evolutionary approaches may survive in the long term if communication between the database and expert system components undergoes radical improvement). In the long term, a revolutionary approach (which starts completely from scratch) is probably necessary for the development of systems that are comparable to relational database management systems.

1.4.5.1 Loose-Coupled Approach

The evolutionary approaches are usually categorised as either 'tight-coupled' or 'loose-coupled' approaches. Loose-coupled systems (such as NAIL! [Ullman 1989]) are regarded as being the easier of the two approaches to implement, because the modifications required to the system components are minimal. Their architecture generally conforms to Figure 1.1.

Figure 1.1 A Loose-Coupled Hybrid Architecture

---

7 This distinction seems to be originally due to [Vassiliou and Jarke 1984]. For some reason, the terms "tight-couple" and "loose-couple" seem to have acquired a number of rather divergent interpretations - the versions used here are an attempt at a synthesis.
In essence, such systems take an existing database manipulation system and an existing artificial intelligence system, and provide a call interface between the two. The artificial intelligence processing component acts as interface and formulates queries which it passes to the database management component; the query is then executed and the result passed back to the artificial intelligence component. In such an approach, the user is definitely aware that there are two distinct system components, as database operations must be performed before logical processing can occur. Thus, the integration is minimal.

Although it is desirable to increase the modelling power available to the user, this must be tempered by the additional requirement that the implementation must still be efficient; but although they are easier to implement than tight-couples, loose-coupled systems - which normally have a Prolog-like language as their artificial intelligence component - tend to be inefficient because there is a mismatch between the declarative paradigm of relational manipulation languages and the procedural aspects of the artificial intelligence component, and this leads to performance problems as the execution of queries tends to be driven by the artificial intelligence component, and so such systems are usually poor at efficiently accessing their database management component. Querying is done in a 'tuple-at-a-time' fashion (using techniques such as proof trees, backtracking, and so on), and this makes little or no use of the 'set-at-a-time' strengths of the database management component - which lie in bulk data transfer and optimisations available over large data sets. Complex logical queries (such as recursion) are particularly badly served by such an approach because, in a memorable phrase from [Mylopoulos and Brodie 1990], they "nickel and dime [the database] to death" by using the naive access methods of the artificial intelligence component. A solution to this problem is probably not possible, given the procedural semantics of Prolog.

1.4.5.2 Tight-Coupled Approach

In direct contrast to this approach, the execution of queries in a tight-coupled system (such as LDL [Tsur 1988; Chimenti et al 1990]) tends to play more to the strengths of the database management component.
The execution of a query is governed by the return of results to a query, not by the query generator. Generally, their architecture conforms to Figure 1.2. The two components are properly integrated and the database manipulation language provides direct support for the logical operations that are normally found in the artificial intelligence component. As Figure 1.2 shows, there may no longer be a difference between the two components, and so any explicit database management component is swallowed up by the artificial intelligence component. This tends to lead to persistent data structures that are directly handled by the artificial intelligence component (such a situation is like that offered by ODE [Agrawal and Gehani 1989], which is a database that offers persistent data structures in a general-purpose programming language).

1.4.5.3 Comparison of Loose-Couples and Tight-Couples

In contrast to loose-couples, tight-couples tend to be based upon the Datalog variation of Prolog ([Ullman 1988, Ceri et al 1989]). Briefly, Datalog is a function-free version of Prolog that takes the relational model as its underlying model, with predicate symbols denoting relations in the same way as the Prolog-using loose-couples do. It also distinguishes between intensional and extensional database predicates in rules but, whereas Datalog has a purely declarative semantics and a comfortable translation into the relational algebra, Prolog semantics are determined in an implementational way - by how the program executes. Tight-couples are generally preferable to loose-couples from a consistency point of view.
As [Stonebraker 1992] has observed, a hybrid system can only guarantee domain consistency if rules and data are closely integrated. Domain consistency does not occur in domains implemented in current relational database implementations because any semantics occur in application programs associated with a domain, and it is possible for the database to be updated without an application program knowing about it - either by another program, or by direct database access (see earlier in this chapter for a demonstration of this problem). Only tight-couples offer such close integration.

Whilst tight-coupled systems are generally perceived to be harder to implement (because knowledge about the execution strategy of the database management component needs to be incorporated into the artificial intelligence component), empirical studies ([Bocca 1989]) have shown that a tight-coupled approach can lead to processing that is at least an order of magnitude faster than that provided by a loose-coupled system. Tight-coupled systems also tend to be more effective because the optimisation strategies of relational database management systems are more useful over large data sets than the optimisations of Prolog-like execution strategies.

1.4.6 Graphical Representations of Logic

Although logic and the programming languages based upon them can be successfully used as principled\(^8\) data manipulation languages - they have a high semantic capture and a more-or-less formal basis in logic - they do not explicitly possess many of the constructs present in the semantic data models. Put simply, logic programming languages are not abstract enough, and are often not completely implementation-free; the user needs to know the number of arguments that a predicate has, which argument is in which position within a predicate, and so on.

This has led to investigations into more graphical forms of logic as data manipulation languages. In a study of various semantic networks (which are graphical representations of logic - such as KL-ONE, KRYPTON, and so on), [Nosek and Roth 1990] have pointed out that semantic networks are much better than predicate calculus for comprehension and ease of

---

\(^8\) There is a formal relationship between the relational algebra and safe nonrecursive Datalog formulas, for example [Ullman 1988].
conceptualisation. Unlike logic programming languages, semantic networks also tend to support object-oriented ideas such as hierarchy and encapsulation.

In their ability to model extensional knowledge, semantic data models such as DAPLEX [Shipman 1981] are very similar to semantic networks in that they are both comprised of networks of entities, relationships between those entities and higher-order taxonomies of entity types [Dayal and Smith 1986].

1.4.6.1 Advantages of a Graphical Notation

Graphical notations have been used for logic since the third century AD, when Porphyry arranged Aristotelian categories into possibly the world’s first semantic network. Even Charles Peirce - one of the earliest inventors of an algebraic notation for predicate logic - was never satisfied with that notation, believing instead that graphs are a more versatile and readable way of showing logical relationships [Sowa 1993]. A system based upon a graphical notation can be as complete as those systems based on logic processing languages and be more readable, and discussion of the limitations of record-based information models by [Kent 1979] contains the interesting remark that models that “are graph-based and based on primitive concepts such as binary relationships and entities ... tend to be more functionally complete in their information processing capability, and more precise in their semantic modelling”.

Relational database designers should be familiar with the benefits of a graphical notation from exposure to entity-relationship diagrams [Chen 1976], which are a convenient way to describe structure and elementary semantics. Although useful for database design, they have little more to offer semantically (see [Carasik et al 1990] for a more detailed discussion). Can a more expressive graphical notation be found?

1.4.7 Conceptual Graph Theory

Although many graphical notations exist, the system of interest in this thesis is that of conceptual graph theory (first referred to in [Sowa 1976], but greatly elaborated upon in [Sowa
1984] and other places - such as [Sowa 1990] - since then). Although a detailed discussion is inappropriate here, it can be said that in their original incarnation in [Sowa 1976], conceptual graphs were intended primarily as a means of describing data and the relationships between such data, and not as a means of storing data. Their advantage lies in the fact that they have naturally developed into a knowledge representation formalism ([Slagle et al 1990] detail expert system requirements, and show that conceptual graphs meet almost all of those requirements).

Conceptual graphs are also being developed - in preference to other semantic networks - as an ANSI normative language for databases by the ANSI X3H4 committee. They are much more general than the entity-relationship model and other semantic data models, can support a direct mapping onto a relational database, and are a kind of ‘object-oriented logic’ ([Hines et al 1990] show a basic match between conceptual graph and object-oriented structures).

1.4.7.1 Subsumption of Other Relational Graphical Notations by Conceptual Graph Theory

The generality of conceptual graphs allow them to subsume other formalisms. For example, NIAM can be subsumed by the ENIAM extension to conceptual graphs [Creasy and Campbell 1992], whilst the entity-relationship model is subsumed by the ECG extension to conceptual graphs [Creasy and Moulin 1989; Creasy and Ellis 1993]. ENIAM interfaces existential graphs with NIAM, giving it the expressive power of first-order logic whilst having explicit constructors for database design. However, it is inferior to conceptual graphs in that it has no means of performing abstraction or generalisation.

1.4.7.2 Conceptual Graph Theory as a Semantic Data Model with Manipulative Component

In addition to their use as a design formalism, conceptual graphs also have a formal system of operations that allow logically sound and consistent manipulation of those structures. These operations are based on a fusion of classical relational operations with ideas from the semantic data models and so conceptual graphs are general enough to be used as a query language. [Sowa 1984] specifies the following advantages and disadvantages of conceptual graphs as a manipulation language:
they require no implementation-specific detail,
- although they do require linguistic sophistication;
+ they place fewer demands on the amount of information that has to be specified by the user,
- but this means that the computer system has to find and insert missing detail;
+ they are general as first-order predicate logic, but they have better mapping to and from natural language.

Although conceptual graphs may not be an ideal manipulation language for the user, they are free from specific implementational considerations.

1.4.8 Use of Conceptual Graphs as a Hybrid Manipulation Language

Conceptual graphs have been successfully used as an expert system formalism (see [Heaton 1994] for the most successful implementation to date), but they can also be used as a powerful artificial intelligence component in a hybrid system. Although this is undoubtedly the case, there have been remarkably few attempts to build such a system. In fact, the only serious implementation known to the author is that of CSL-0 [Poesio 1987b], which is a hybrid system built upon a binary relational storage structure (see Chapter 3 for a discussion of this issue). None of the systems mentioned above have serious implementations, and many of them also contain structures that explicitly alert the user to the fact that a database is being used.\(^9\)

Conversely, a formal approach to the problem is elaborated in the next chapter - an approach that requires no special structures, and only contains extensions that, whilst making conceptual graphs relationally complete, may also be seen as principled extensions regardless of whether a database is being used or not.

\(^9\) The most obvious case of this is in [Boksenbaum et al 1993], where the idea of the relational relationship is introduced. This special-purpose concept is a very explicit first-order version of a relation scheme, with attached concepts being the attributes of that relation scheme. Although translations into relational algebra are simple from this system, the notion of the relational relationship is very doubtful from the point of view of formal logic, being a special-purpose node with no meaning outside the mapping between the relational model and the form of conceptual graphs extended in this way.
1.5 Summary

The relational model is so popular because it simply models a large class of business and other application domains, and it has developed over the years into the pre- eminent data storage and retrieval system available. However, it is also the case that a growing class of application domains - generally those that require complex structures and powerful operations over them - is emerging, and the relational model is ill-equipped to deal with this class of domains.

These deficiencies have led to the development of a large number of new approaches - most of which have greater semantic capture than the relational model, but few of which have developed a tractable set of manipulations over the structures they propose. The sheer volume of ill-formed approaches that have appeared since it became clear that the relational model was limited simply shows that the relational model - which is well-formed and logically sound - is worth preserving. This perception has given rise to loose-coupled hybrid systems\(^\text{10}\), which uses the relational model for efficient storage and retrieval, whilst using a semantically rich language to handle the semantics of the stored data. Systems based on the logic programming language Prolog (and its more database-oriented subset Datalog) are in this class; such languages have high semantic capture and a well-established correspondence with the operators of the relational model, and also have a number of well-developed implementations.

Such systems are semantically rich (as a result of their correspondence with first-order predicate logic), yet also map to structures in the relational model. Other systems with hybrid capability include graph-based formalisms such as EER, NIAM, and conceptual graph theory; traditional entity-relationship diagrams can be regarded as more or less isomorphic with the relational model as regards their semantic content, and so do not include them within this area. Conceptual graph theory is perhaps the most interesting of these formalisms, in that they are a development from both relational databases and expert systems (whereas other systems tend to proceed from just one extreme). They are also perhaps the most powerful and expressive of the graph-based

---

\(^{10}\) Some tight-coupled hybrid systems - such as LDL [Chimenti et al 1990] and GRAS [Kiesel et al 1995] - have moved beyond this, offering high semantic capture in an integrated storage environment. However, the interest here is primarily with loose-coupled systems.
formalisms currently available, being able to subsume EER and NIAM as well as entity-relationship diagrams; they also have a well-developed manipulative component, which the others do not. It is significant that they are also being proposed as the X3H4 ANSI standard language for database definition and manipulation.

1.6 Research Aims

The main aim of this chapter has been to show that the relational model is limited in what it can represent and process, and that a logic-based extension to the model represents a tractable approach to increasing its expressive power.

1.6.1 Primary Aim: Conceptual Graphs as a Conceptual Schema Definition Language

The primary aim in the following work is to show that conceptual graphs can act as an implementation-independent conceptual schema definition language for an implementation of the relational model.

1.6.2 Aim #2: Correspondence Between Conceptual Graphs and the Relational Model

The second aim is to find out how close a correspondence conceptual graph theory has with the logic of the relational model, and to find out if conceptual graph theory is formally more expressive than relational manipulation languages - and, if so, where. If it is found that if there are parts of conceptual graph theory that possess no mapping to the relational model (and vice versa), conceptual graph theory will be modified in order to remove these deficiencies.

1.6.3 Aim #3: Use of Extended Conceptual Graphs in a Hybrid System

Given the trend towards hybridisation, can such an extended conceptual graph theory can be used as the theoretical basis for the construction of a hybrid system based on conceptual graphs and a relational database management system? Such a hybrid implementation of conceptual
graphs and the structures and operations of the relational model has not yet been demonstrated (in fact, the only attempt at such a system appears to be the CSL-O system [Poesio 1987b]).

1.6.4 Aim #4: Increased Declarativeness of a Hybrid System

Given that the declarativeness of relational languages is insufficient for easy use by end-users, any system constructed should require no implementation-specific knowledge of an application domain on the part of the users - ideally, the system should be so declarative that the user would not be able to tell if a database system was being used for storage and retrieval of data. In order to achieve this, any system constructed will have to take responsibility for the design and maintenance of storage structures, as well as being responsible for creation and maintenance of mappings between its two hybrid levels.

1.6.5 Aim #5: Use of Existing Technology in Hybrid System Construction

Low-level code manipulation of existing database packages and the design of a new storage system from scratch are outside the remit of this research, so the database management component of any hybrid system constructed should be achieved within the framework of an existing commercial database package - although this almost inevitably means that any system constructed will be more loose-coupled than tight-coupled. However, it will have the advantage that any system developed can be applied to existing relational database systems, and will also give us painless access to many man-years of database development.

1.6.6 Aim #6: Investigation of Legacy Database Reuse in a Hybrid System

Finally, can existing legacy databases can have their semantics increased by mapping into any conceptual graph - relational database hybrid implementation developed? In order to achieve this, it will be necessary to devise a formal system for mapping existing relational structures - which contain very little semantic information - into the concepts and relations of conceptual graphs.
1.7 Organisation of Thesis

Following this introductory chapter are:

- Chapter 2, which deals with the theoretical basis of the relational model and how conceptual graph theory can (and occasionally cannot) be related to it. This chapter is the formal basis of a hybrid correspondence between a commercial relational database management system and a semantically rich interface layer based on an extended version of conceptual graphs outlined here.

- Chapter 3, which describes the design of that semantically rich layer in C-GRASS, or the Conceptual Graph Relational Adaptive Storage System. This chapter contains a general description, followed by a discussion of how both the relational database management component and the semantic layer interact, and what problems arise from that interaction. It also includes details of how the extensions to conceptual graph theory proposed in the previous chapter have been implemented in practice, and also gives details of an *adaptive* property of the system that is thought to be unique, and allows the underlying database structures to change as the application domain changes.

- Chapter 4, which outlines the implementation of the design of C-GRASS.

- Chapter 5, which evaluates the correctness and efficiency of some key operations of C-GRASS, as well as giving some indicators towards the correctness and efficiency of the overall system.

- Chapter 6, which concludes the thesis and outlines possible future extensions to the work contained therein.
CHAPTER 2
Conceptual Graphs and the Relational Model

There are countless applications that use relational database technology for storage and manipulation of data, and conceptual graph theory seems to be an ideal formalism for extending their semantic capture and general expressive power. However, the early versions of conceptual graph theory that were concerned with mappings to the relational model [Sowa 1976] are informal, and are incomplete in many areas. A formal theoretical mapping between conceptual graph theory and the relational model is therefore the objective of this chapter.

2.1 Chapter Objectives

This chapter aims to formalise and extend conceptual graph theory in order that a correspondence between its intensional structures and operations and the extensional structures and operations of the relational model can be shown. Formal relationships between the structures of the two models must be shown, along with extensions to conceptual graph operations that are necessary in order to make a correspondence between the models. With a number of modifications and extensions, the chapter aims to show that conceptual graphs subsume the relational model, whilst still possessing markedly more functionality and so greater modelling power.

This chapter also aims to show that conceptual graphs have at least the expressive power of the relational model; to do this, it must be shown that conceptual graph theory has expressive power that is at least equal to that of the relational algebra [Codd 1970], which is the fundamental basis of all relational manipulation languages. An exposition of both the relational algebra and the tuple relational calculus must be given and a formal correspondence between conceptual graphs and the relational algebra must be demonstrated by showing that a formal
correspondence between conceptual graphs and the tuple relational calculus [Codd 1972] exists. If such correspondences can be demonstrated and as tuple relation calculus is known to be relationally complete [Ullman 1988], conceptual graphs will therefore be shown to be relationally complete (and usable as a relational manipulation language).

2.2 Correspondence Between Relational Structures and Graph Structures

Before it can be shown how conceptual graph theory corresponds to the relational model at a structural level, it is necessary to understand, at a relatively formal level, what structures conceptual graphs actually need to correspond to. Put simply, the relational model is just a particular way of representing and manipulating data - in this case, one that is based upon the mathematical theory of relations. Unlike earlier data models, it is value-oriented rather than object-oriented; and unlike earlier models (which were generally object-oriented in approach), this value-oriented property allows the formation of a manipulation language that is closed [Gray 1985].

2.2.1 Formal Structural Description of the Relational Model

It is necessary to define a little terminology before proceeding further.

Definition 2.1 Domain of a Relation: a domain is a non-empty set of values. For example, the following sets are domains:

\[
\begin{align*}
\text{COUNTRY} & = \{\text{Great Britain, Peru, Japan}\} \\
\text{CAPITAL} & = \{\text{London, Lima, Tokyo}\} \\
\text{CONTINENT} & = \{\text{Europe, Asia, South America}\}.
\end{align*}
\]

1 This section may be omitted by those familiar with the relational model, as it simply outlines the most basic concepts that underlie it, and provides a formal basis for talking about the relational calculus.
Definition 2.2 Cartesian Product: the Cartesian product of a set of domains $D_1$, $D_2$, ..., $D_n$ (normally written as $D_1 \times D_2 \times ... \times D_n$) is the set of $n$-tuples $(d_1, d_2, ..., d_n)$ such that $d_1$ is an element of the domain $D_1$, $d_2$ is an element of the domain $D_2$, and so on. For example, the product of $COUNTRY \times CAPITAL \times CONTINENT$ is the set of 3-tuples (triples):

$\text{(Great Britain, London, Europe),}$
$\text{(Great Britain, Lima, Europe),}$
$\text{(Great Britain, Tokyo, Europe),}$
$\text{...}$
$\text{(Japan, London, South America),}$
$\text{(Japan, Lima, South America),}$
$\text{(Japan, Tokyo, South America),}$

Definition 2.3 Relation (and Relation Scheme): a relation is a meaningful subset of the Cartesian product of one or more domains. The set of data descriptors (or attribute names) associated with each domain in a relation are often referred to as the relation scheme for that relation. For example, the above Cartesian product for the domains $COUNTRY \times CAPITAL \times CONTINENT$ contains the meaningful subset:

$\text{(Great Britain, London, Europe),}$
$\text{(Peru, Lima, South America),}$
$\text{(Japan, Tokyo, Asia),}$

and $(COUNTRY \ CAPITAl \ CONTINENT)$ is the relation scheme.

Note that a relation may be empty and still be meaningful. Relations are often expressed in a tabular form:
where each row of the table is an n-tuple of the relation. Within such a table, the ordering of the rows is unimportant. The ordering of columns is significant however, as two columns may have identical attribute names (and thus map to the same domain) but mean different things within the relation. A classical example of this is the representation of part-subpart relationships in a table:

<table>
<thead>
<tr>
<th>PART</th>
<th>PART</th>
</tr>
</thead>
<tbody>
<tr>
<td>A123</td>
<td>B567</td>
</tr>
<tr>
<td>A123</td>
<td>B678</td>
</tr>
<tr>
<td>B890</td>
<td>C456</td>
</tr>
<tr>
<td>B890</td>
<td>C567</td>
</tr>
</tbody>
</table>

where the second column represents the parts needed to make the part in the first column. Thus, column ordering may carry some semantic information and has to be preserved.

**Definition 2.4** $dom(X)$: the domain of a given attribute name is mapped to by this function. For example, $dom(COUNTRY) = \{Great Britain, Peru, Japan\}$.

**Definition 2.5** Tuple: tuples are simply the elements of a relation. For example, the geographical relation of Definition 2.3 has three tuples.

**Definition 2.6** Arity (alternatively Degree): the arity of a relation scheme is the number of domains in that scheme. The geographical relation of Definition 2.3 has arity 3 (or is 3-ary) because it maps to three domains; this said however, the Cartesian product $PERSON \times PERSON$ has arity 2 because each use of the $PERSON$ domain is distinct.
Definition 2.7 Component: a tuple of arity $n$ has $n$ components.

Definition 2.8 Cardinality: the cardinality of a relation scheme is the number of tuples in that scheme. For example, the geographical relation of Definition 2.3 has a cardinality of 3. Although relations may theoretically be infinite, practical necessities of storage impose finite limitations on the arity and cardinality of relations, and so both of these properties possess a definite value.

Definition 2.9 Primary Key: a combination of attributes that are unique for each tuple. The whole relation scheme may form the key in some cases.

The only structure upon which the model is built is the relation; there are no other structures present. Each relation can be thought of as consisting of a relation scheme and a series of tuples that conform to that relation scheme both in arity and in terms of $\text{dom}(A)$, where $A$ is an attribute name of the relation scheme. The relation scheme is time-invariant, whereas the set of tuples is time-variable (so the degree of a relation is fixed, whilst its cardinality may vary).

Within a relation, tuples may not be duplicated (as the relation is a set and duplicates in a set are redundant; the use of a primary key guarantees this property), and their order is not significant (once again, as in set theory). The order of attributes is also not significant, in theory at least (although see the earlier terminological description of relations for practical objections to this position). Finally, all simple attribute values must be atomic.

2.2.2 Mapping of Conceptual Graphs to Relational Structures

Conceptual graphs are bipartite structures consisting of atomic concepts linked together by directed conceptual relations. Obviously, there is a simple relationship between instantiated atomic concepts and a value from a given domain that corresponds to the type of the concept. Relation schemes also broadly correspond to the basis type of conceptual graph theory, being a descriptor for a set of records; the process of individuation maps the type to the storage
structures [Sowa 1984]. Although conceptual graph theory originally developed from work into developing a more user-friendly interface to relational applications, conceptual graph theory still has no really formal methodology beyond these simple correspondences for mapping to relational structures, and these must be formalised in some way. It is therefore necessary to describe a method by which the concepts and conceptual relations of conceptual graph theory can share some common ground with the attributes and relations of the relational model, and the following simple additions to the structure of conceptual graphs enable such a mapping to be drawn between the two structural systems:

**Definition 2.10 Conformity and Role Domain Mappings:** if a concept $c$ in a conceptual graph $u$ participates in no conceptual relations, then $c$ possesses a *conformity domain mapping* $\mu_c$ which maps $c$ to the single attribute name $n$ of a 1-ary relation scheme $s$ that acts as the conformity relation for the concept label of $c$; $c$ is therefore solely associated with the mapping $(s,n)$. If a concept $c$ in a conceptual graph $u$ participates in one or more conceptual relations $\{R\}$, then for each $r$ which is an element of $R$, $c$ possesses one or more *role domain mappings* $\mu_R$ that maps $c$ to an attribute name $n$ in a relation scheme $s$; $c$ is therefore associated with a list of mappings $(s,n), \ldots$. In short, $\mu_R$ maps an intensional $c$ to an extensional domain.

Within $u$, any $\mu_R$ does not have to be unique, and the role domain mappings $\mu_R$ of any $c$ may vary over time. The reason for this is linked to the issue of schema evolution and storage adaptation, which shall be explored later. It should be noted, though, that the conformity domain mapping $\mu_c$ of any $c$ is time-invariant - as long as $c$ participates in no conceptual relations, it will possess $\mu_c$; but if this state of affairs changes, then $c$ will lose $\mu_c$ and gain one or more $\mu_R$. Although such domain mappings are a purely semantic property of conceptual graphs and do not require any additional syntax to deal with them, there is an obvious need to be able to represent them in order that they can be discussed. For the *purpose of exposition only*, Figure 2.1 shows a graphical means of displaying domain mappings in the display form. The heavy circle surrounding the conceptual graph represents a relation scheme $R$; conventionally, the name $R$ is attached to the exterior of this circle.
Figure 2.1 Showing Domain Mappings in the Display Form

Associated with each concept $k$ inside the circle is a domain $C_k$; this is sited as closely to the concept as possible, whilst being attached to the circle representing $R$. Thus, this graph represents the relation scheme $R (C_1 \ C_2)$.

In the larger example represented by Figure 2.2, multiple relation schemes $R_1$, $R_2$, $R_3$ and $R_4$ exist, with multiple role domain mappings being possessed by a number of concepts - [PERSON] appears as both the domains $R_1.C_1$ and as $R_3.C_1$, [JOB] appears as both the domains $R_1.C_2$ and $R_2.C_1$, and [DEPARTMENT] appears as both the domains $R_3.C_2$ and $R_4.C_1$.

Note that the domains associated with these concepts are attached to their own relation - in these diagrams, a concept may not match to more than one domain in each relation$^2$. The sites of these multiple role domain mappings tend to indicate join locations between the various relation schemes.

For want of a better term, conceptual graphs possessing the property of domain mappings are referred to as domain-extended conceptual graphs.

$^2$ Away from the diagramming conventions, a concept may possess identical elements of $\mu_R$. For example, [PERSON] would possess the mapping $R_3.C_1$ three times - once for each dyad in the $R_3$ subgraph in which it participates.
**Figure 2.2 Domain Mappings in a Simple Conceptual Graph**

**Definition 2.11** *Domain-Extended Conceptual Graph*: a domain-extended conceptual graph \( u \) is a conceptual graph that possesses conformity and role domain mappings for all concepts \( c \) in \( u \), in addition to any other properties.

Explanation of the mechanism that assigns these mappings is deferred for the moment. This approach is preferable to that outlined in [Sowa 1976], which is rather naive.
Figure 2.3 Hypergraph of Figure 2.2

It also removes the monopoly of mappings between the two models from the basis type\textsuperscript{3}.

2.2.3 Similarity of Domain-Extended Conceptual Graphs to Hypergraphs

Conceptual graphs that have been extended in this way are obviously related to hypergraphs [Maier 1983]. The schemes mapped to by $\mu_R$ from a conceptual graph $u$ can be viewed as hyperedges - that is, nonempty sets of nodes - of a hypergraph that corresponds to $u$. Such hypergraphs are always reduced, as no hyperedge (subgraph) will ever properly contain another. As an example of this, consider Figure 2.3, which is the hypergraph form of the domain-extended conceptual graph of Figure 2.2.

2.2.3.1 Datalog and Hypergraphs

In Datalog, the hyperedges of such hypergraphs are extensional predicates; such predicates represent whole relation schemes (and thus represent tuple variables, as shall be seen in Section 2.3.3). The drawback with this approach is that the user still needs a certain level of knowledge about the organisation of those predicates in order to form proper assertions and inquiries -

\textsuperscript{3} More precisely, a basis type is similar to the relational view mechanism, in that a basis type encapsulates domains of a relation scheme in a way analogous to a relational view using only certain domains of one or more relation schemes.
conversely, using domain-extended conceptual graphs instead of such explicitly extensional predicates maintains a higher level of abstraction, and so hides information about the organisation and nature of those extensional predicates by use of $\mu_R$. This is an advantage of domain-extended conceptual graphs over lower-level approaches such as Datalog, and is especially advantageous when considering the aim of shielding the user from implementation-specific knowledge.

2.2.4 Domain-Extended Conceptual Graphs as Window Functions

Because conceptual graphs have a high level of abstraction, relational database inquiries formed via the use of $\mu_R$ also have a number of close parallels to the window functions of the universal relation hypothesis (see [Vardi 1988] for a useful exposition of the hypothesis). As [Ullman 1989] has noted, the need for some form of universal relation model is essential if natural language processing of a database is ever to become possible - for the simple reason that the universal relation model is a level of abstraction higher than the relational algebra and associated languages.

2.2.5 Advantages of a Universal Relation Approach

To see the advantage of a universal relation approach, consider the investment house example of Appendix E. In this example, the statement required to find the investors handled by the broker #100, as well as their shares and their quantity, was:

\[
\begin{align*}
\text{RETRIEVE} & \ (\text{FIRST.INVESTOR#}, \ \text{FIRST.STOCK}, \ \text{FIRST.QUANTITY}) \\
& \text{FROM SIQ FIRST, IB SECOND} \\
& \text{WHERE SECOND.BROKER#} = 100 \\
& \text{AND SECOND.INVESTOR#} = \text{FIRST.INVESTOR#}
\end{align*}
\]

where the user is required to know the names of the tables and how they link together - in the previous chapter, this was referred to this as implementation-specific knowledge, and was cited as a deficiency of the relational model.

47
However, it can now be seen that it is not the model *per se* that is at fault, merely the languages that manipulate it.

The universal relation model attempts to increase the declarativeness of those languages by removing some of the implementation-specific knowledge required - the location in a relation scheme of a given attribute, and how relation schemes must be joined to satisfy a particular query. In the syntax of System/U [Ullman 1989], the above query would be expressed as:

```sql
SELECT (INVESTOR#, STOCK, QUANTITY)
WHERE BROKER# = 100;
```

This is much simpler for potential end-users to understand and to formulate, because the hard work is being done behind the scenes by the computation of window functions. For the above query, the window function in Figure 2.4 computes the selections and joins behind the scenes, in order to retrieve the attributes required by the query without the user having to specify them.

### 2.2.6 Null Information in the Relational Model and Conceptual Graphs

Null information has always been a problem in relational databases, and using conceptual graphs as a hybrid interface language between the user and relational storage doesn't necessarily remove the problem.
[COMPANY: #1] -
(MANF)←[PRODUCT: #2] -
(COLR)→[COLOUR: #4] -
(NAME)→[COLNAME: “Blue”],
(COLR)→[COLOUR: #5] -
(NAME)→[COLNAME: “Grey”],
(MANF)←[PRODUCT: #3] -
(COLR)→[COLOUR: #6] -
(NAME)→[COLNAME: “Red”],
(COLR)→[COLOUR: #7] -
(NAME)→[COLNAME: “Pink”].

Figure 2.5 A Fully Instantiated Conceptual Graph

Essentially, the problem relates to the status of generic concepts in storage. [Sowa 1984] states that generic concepts correspond to null values - but what does a null value actually mean in the context of storing non-types? For example, consider Figure 2.5.

In this example, it might be that a company manufactures many products, and that a product could be multicoloured (and of course, many products could share colours). The relation schemes required to store this graph might be:

\[
\lambda_1 \text{(PRODUCT} \text{COMPANY)} \\
\lambda_2 \text{(PRODUCT} \text{COLOUR)} \\
\lambda_3 \text{(COLOUR} \text{COLOURNAME)}
\]

where \(\lambda_1\) enforces that a company can make multiple products, and that a product can be manufactured by many producer companies; \(\lambda_2\) enforces that a product may have multiple instances of colour, but each such instance only belongs to one product; and \(\lambda_3\) enforces that
any instance of colour can only be one colour, but that a given colour can be used in multiple instances.

The graph in Figure 2.5 would therefore become the tuples:

\[
\begin{align*}
\lambda_1 & \quad (#2, #1) \\
& \quad (#3, #1) \\
\lambda_2 & \quad (#2, #4) \\
& \quad (#2, #5) \\
& \quad (#3, #6) \\
& \quad (#3, #7) \\
\lambda_3 & \quad (#4, \text{Blue}) \\
& \quad (#5, \text{Grey}) \\
& \quad (#6, \text{Red}) \\
& \quad (#7, \text{Pink})
\end{align*}
\]

However, what happens if the graph in Figure 2.6 was asserted, where PRODUCT is a generic concept? [Sowa 1984] suggests that this would be represented as a null value, and so the extra tuples needed to store it in the above schemes would be:

\[
\begin{align*}
\lambda_1 & \quad (\text{null, } #1) \\
\lambda_2 & \quad (\text{null, } #4) \\
& \quad (\text{null, } #7)
\end{align*}
\]

Because the information making up this graph is now dispersed over multiple relation schemes, it is now impossible to reconstruct the original graph - because the null values has no information about how to do this. It is therefore necessary to generate unique internal surrogates, such as:

\[
\begin{align*}
\lambda_1 & \quad (*1617, #1) \\
\lambda_2 & \quad (*1617, #4) \\
& \quad (*1617, #7)
\end{align*}
\]

in order to perform reconstruction of conceptual graphs - that are not fully instantiated - from storage.
Just as dependencies can be used to create and maintain efficient storage structures, they can also be used to handle null information as well [Maier 1983]. The generation of unique internal surrogates can be guided by the cardinality and membership information that is extractable from the application domain. For example, if it is known from extracted information that every person must have a single national insurance number, then any assertion of any information about an instantiated PERSON - such as the graph:

[PERSON: #80089]

(LOC)→[DEPARTMENT: #88].

must be tested to see if that person has a known national insurance number; if not, then a surrogate is allocated, and so the graph effectively becomes:

[PERSON: #80089] -

(LOC)→[DEPARTMENT: #88]

(ATTR)→[NI-NUMBER: *1618].

when being stored; this prevents the storage of nulls and the situation described above.

Figure 2.6 A Conceptual Graph That is Not Fully Instantiated
2.3 Correspondence Between Relational Operations and Graph Operations

Having formalised the structural correspondence between the relational model and conceptual graph theory, the next step is to look at two manipulation systems which operate over those structures - as was mentioned in the Introduction, structure without corresponding operators is “rather like anatomy without physiology” [Codd 1979].

2.3.1 Description of the Relational Algebra

The relational algebra is the fundamental system by which relations are manipulated, and is a highly mechanical and prescriptive system. Conversely, the tuple relational calculus is much more of a descriptive system - it allows the user to describe what is wanted, rather than how to get it.

**Definition 2.12 Operators of the Relational Algebra:** [Codd 1970] defined eight operators as making up the relational algebra:

i \( \cup \) (Union) takes two relations of similar degree \( X \) and \( Y \), and produces a relation \( Z \) which contains the tuples of both \( X \) and \( Y \).

ii \( \setminus \) (Set Difference) takes two relations of similar degree \( X \) and \( Y \), and produces a relation \( Z \) which contains all the tuples of \( X \) that aren’t also present in \( Y \).

iii \( \times \) (Cartesian Product) takes two relations \( X \) and \( Y \) (of degree \( x \) and \( y \) respectively and cardinality \( a \) and \( b \) respectively), and produces a relation \( Z \) (of degree \( x + y \)), where the first \( x \) components of tuple in \( Z \) correspond to a tuple in \( X \) while the remaining \( y \) components correspond to a tuple in \( Y \) (\( Z \) thus has a cardinality that is the product of \( a \) and \( b \)).

iv \( \pi \) (Project) takes a relation \( X \) and produces a relation \( Y \) which has only some of the attributes of \( X \) (and so has a smaller degree than \( X \)).
\[ \sigma \] (Restrict or Select) takes a relation \( X \) and produces a relation \( Y \) which contains only some of the tuples of \( X \) (and so has smaller cardinality than \( X \));

These five operations are primitive; the following three operations can be constructed from a combination of primitives:

vi \( \ast \) (Theta-Join) takes two relations \( X \) (which contains an attribute \( A \)) and \( Y \) (which contain an attribute \( B \)) and produces a relation \( Z \) which is the Cartesian Product of \( X \) and \( Y \), with the restriction \( a \theta b \), where \( \theta \) is an arithmetic comparator (=, \geq, etc) and \( a \) and \( b \) are elements of \( \text{dom}(A) \) and \( \text{dom}(B) \) respectively. If \( \theta \) is the equality comparator, then the join is an equijoin.

vii \( \cap \) (Intersection) takes two relations of similar degree \( X \) and \( Y \), and produces a relation \( Z \) which contains the tuples present in both \( X \) and \( Y \).

viii \( + \) (Divide or Quotient) takes two relations \( X \) and \( Y \) (of degree \( x \) and \( y \) respectively, and where \( x > y \)), and produces a relation \( Z \) (of degree \( x - y \)) of tuples such that for all tuples \( (a_1, \ldots, a_{x-y}) \) in \( Y \), the tuple \( (a_1, \ldots, a_x) \) is in \( X \). \( X \) is normally binary and \( Y \) unary in this operation - which, not surprisingly, is little used.

The relational algebra is the purely mechanical means by which relational structures are manipulated; if a relational manipulation language cannot show a correspondence to the relational algebra, it cannot be considered as a proper relational manipulation language.

2.3.2 Why the Relational Algebra is Limited: Problems of General-Purpose Languages

Although the relational algebra may seem rather limited in what can do, it is limited for very good reasons. If a completely general-purpose programming language were to be used as a relational manipulation language, the number of cases where an advantage would be gained by this approach would be heavily outweighed by the number of cases where this approach would
be disadvantageous. As [Ullman 1988] has noted, the use of arbitrary programs as queries would require the user to know everything about the physical data structures used - which immediately divorces the user from many of the advantages of the relational model; in addition, arbitrary programs would need to be geared to their own data structures - which immediately brings the important algebraic property of optimisation into doubt.

For example, consider the use of Prolog as a relational manipulation language. A number of hybrid systems between Prolog and relational databases have been constructed, but the Prolog part of these systems is often linked to the relational structures by means of special-purpose predicates. The result is that, while Prolog has the power of a general-purpose programming language, query optimisation is impossible as queries are guided by the Prolog execution model.

In short, the use of a general-purpose programming language would take the user away from the logical view of data, and would not be able to tune its operations. The retention of both of these properties is the primary reason why the relational algebra is limited in the way it is - it can solve the optimisation problem, whilst still having sufficient expressive power to be useful. Some models have recognised the importance of this - most notably the Datalog version of Prolog, which is explicitly geared to operating over relational structures whilst possessing most of the functionality of Prolog.

So, although there are areas where conceptual graphs do not map into the relational model, any hybrid conceptual graph processor should therefore make as much use of the relational algebra as possible where conceptual graphs logically correspond to the relational model, in order to retain the optimisation strategies developed for implementations of the relational model.

2.3.3 Formal Specification of the Tuple Relational Calculus

Given that relational algebra is relationally complete, can it be shown that conceptual graph theory is capable of performing all operations that relational algebra is capable of performing? Whilst a direct translation between the two formalisms is possible, it is easier to show a formal correspondence between conceptual graph theory and the tuple relational calculus, which is
known to be convertible into relational algebra and is thus relationally complete [Date 1990; Ullman 1988; Maier 1983]. The tuple relational calculus is a subset of the first-order predicate calculus, and is the basis for almost all commercial relational database languages. Its basic construct is the tuple variable.

**Definition 2.13 Tuple Variable [Codd 1972; Maier 1983]:** a tuple variable \( t \) is a variable whose permissible values are tuples of the relation scheme that it ranges over. For example:

\[
geog(x)
\]

is a tuple variable that ranges over the geographical scheme given in Definition 2.3. If \( geog \) is a legitimate relation scheme then, at any given time, \( geog(x) \) represents some tuple of \( geog \). For a given application domain, \( t \) can be constructed from:

i. a universe of attributes \( U \). For each attribute \( A \) which is an element of \( U \), \( \text{dom}(A) \) maps to a set of values.

ii. a set of binary comparison operators \( \Theta \) to operate over those domains.

iii. a set of relation names \( \{r_1, r_2, \ldots, r_n\} \) on the relation schemes \( R_1, R_2, \ldots, R_n \). All elements of \( R_1, R_2, \ldots, R_n \) must be elements of \( U \).

Instead of just standing for a single value (as is the case in regular logic), a tuple variable stands for a set of values. Formation of such variables is dependent on the application domain (as seen in the above definition), and by the following rules:

**Definition 2.14 Construction of Atomic Formulas:** tuple variables can be combined into atomic formulas by the following rules:

i. for any known relation name \( r \) and for any tuple variable \( x \), \( r(x) \) is an atomic formula where \( x \) is a member tuple of \( r \).
for tuple variables x and y (which can be the same), a binary comparator \( \theta \)
(which is an element of \( \Theta \)), and attributes A and B that are elements of U
and can be compared by \( \theta \), \( x(A) \ \theta \ y(B) \) is an atomic formula.

for any tuple variable x, a binary comparator \( \theta \) (which is an element of \( \Theta \)),
and attributes A and B that are elements of U and can be compared by \( \theta \), if
c is a constant in \( \text{dom}(A) \) then \( c \ \theta \ x(B) \) is an atomic formula; if c is a
constant in \( \text{dom}(B) \) then \( x(A) \ \theta \ c \) is an atomic formula.

As in classical first-order logic, atomic formulas can be connected together into more complex
formulas by the use of logical connectives.

**Definition 2.15 Construction of Complex Formulas:** given a collection of tuple
variables t and the set of connectives \{ \neg, \land, \lor, \exists, \forall \}:

i any atomic formula is a formula.

ii if \( f \) is a formula, so is \( \neg f \).

iii if \( f \) and \( g \) are formulas, so are \( f \land g \) and \( f \lor g \).

iv if \( f \) is a formula, so is \( \exists x(R)f \), where \( R \) are attributes in U and \( x \) is a tuple
variable. If there is a tuple in \( R \) that makes \( f \) true when that tuple is
substituted for \( x \), then \( \exists x(R)f \) is true.

v if \( f \) is a formula, so is \( \forall x(R)f \), where \( R \) are attributes in U and \( x \) is a tuple
variable. If every tuple in \( R \) makes \( f \) true when that tuple is substituted for
\( x \), then \( \forall x(R)f \) is true.

vi if \( f \) is a formula, so is \( (f) \).

vii connectives have the following precedence when determining scope of
parentheses: \( \{ \exists, \forall \}, \neg, \land, \land \) and \( \lor, \exists \) and \( \forall \) have equal precedence.

2.3.3.1 Examples of Tuple Relational Calculus Formulas

A few examples should help to illustrate the rules of construction given in the previous three
definitions:
\[ \text{geog}(x) \land x(\text{CAPITAL}) = \text{"Paris"} \]

or 'tuple variable \( x \) is a member of \( \text{geog} \), and \( \text{CAPITAL} \) in that tuple is "Paris";

\[ \text{geog}(x) \land \neg x(\text{CONTINENT}) = \text{"Europe"} \]

or 'tuple variable \( x \) is a member of \( \text{geog} \), and it is false that the continent in that tuple is equal to "Europe";

\[ \exists x(\text{COUNTRY CAPITAL CONTINENT}) \ (\text{geog}(x) \land x(\text{CAPITAL}) = \text{"Paris"}) \]

or 'there exists a tuple variable \( x, x \) is a member of \( \text{geog} \), and \( \text{CAPITAL} \) in that tuple is equal to "Paris".

2.3.3.2 Requirements for Construction of Legal Tuple Relational Calculus Formulas

Not all of the formulas that can be constructed from these components are legal. For example, it is possible to construct illegal formulas like:

\[ \text{geog}(x) \land x(\text{BOOKTITLE}) = \text{"Hard Times"} \]

(2.1)

where the tuple variable \( x \) is associated with the relation scheme \( \text{geog} \), but \( \text{BOOKTITLE} \) is not an attribute of \( \text{geog} \). Legal formulas in the tuple relational calculus depend on three properties - binding within formulas, typing of tuple variables, and mentions of variable attributes.

The binding of variables within formulas has to be strictly controlled in order to avoid confusion when evaluating the formula. Except for atomic formulas (in which all variables are necessarily free, as there is no quantification), all formulas may contain quantifiers that bind tuple variables. If a formula contains a tuple variable \( x \) which is unquantified, then another formula may bind it. For example, \( x \) is a variable that is free in the formula \( f \). Another formula \( g \) may now come along and bind \( x \):
\[ g = \exists x(R)f \quad \text{and} \quad g = \forall x(R)f \]

If another formula \( h \) tries to bind \( x \) in \( g \):

\[ h = \exists x(R)g \quad \text{and} \quad h = \forall x(R)g \]

then \( g \) would not be legal as \( x \) is previously bound in \( f \).

Within a formula, a tuple variable is 'typed' to a single relation scheme only - for example: if \( \text{geog}(x) \), then it can be said that \( x \) is typed to \( \text{geog} \). All 'mentions' of attributes in \( x \) must belong to the type of \( x \). Bearing this in mind, some typing and mention conditions can be imposed on the atoms and formulas of the tuple relational calculus.

**Definition 2.16** Typing and Mention Legality Conditions: given a tuple calculus formula \( f \), then:

1. if \( f \) is the atomic formula \( r(x) \) where \( r \) is a relation scheme, the type of \( x \) must be \( r \) and all mentions of \( x \) must be attributes of \( r \).
2. if \( f \) is the atomic formula \( x(A) \land y(B) \), then the types of \( A \) and \( B \) are undefined (as they are attributes, not relation schemes), all mentions of \( x \) must be the attribute \( A \), and all mentions of \( y \) must be the attribute \( B \).
3. if \( f \) is the atomic formula \( c \land x(A) \) or \( x(A) \land c \), then the type of \( x \) is again undefined (attributes again) and all mentions of \( x \) must be the attribute \( A \).
4. if \( f \) is the formula \( \neg g \), then for every tuple variable \( x \) appearing free in \( f \), the type of \( x \) in \( f \) must be equal to its typing in \( g \); similarly, the mentions of \( x \) in \( f \) must be equal to the mentions in \( g \).
5. if \( f \) is the formula \( g \land h \) or \( g \lor h \), then for any tuple variable \( x \) appearing free in \( f \), the type of \( x \) in \( g \) must be the same as that of \( x \) in \( h \), and the mentions of \( x \) in \( f \) must be the same as the union of the mentions of \( x \) in \( g \) and \( h \).
if \( f \) is the formula \( \exists x(R)g \) or \( \forall x(R)g \), then the type of \( x \) must be \( R \), and the mentions of \( g \) must be a subset of \( R \). As \( x \) is not free in \( f \), it has no type or mentions in \( f \) (although any other tuple variables in \( f \) that are also in \( g \) must have the same type and mentions in both formulas).

If any of these three properties - binding, typing or mentions - are violated, then the formula is not legal. This is why formula (2.1) is not legal - it violates the mentions property.

2.3.3.3 Formation of Tuple Relational Calculus Expressions

Once all formulas are legal, they can be used to form expressions that can be used as queries over relations; such expressions take the form:

\[
\{ x(R) \mid f(x) \}
\]

where:

- \( f \) is a legal formula, as outlined previously;
- \( x \) is the only tuple variable that is free in \( f \);
- all attributes of \( R \) can be found in the universe of attributes \( U \).

Essentially, \( x(R) \) is the result of the expression - it may be all or part of a known relation scheme, or it may be a collection of attributes from various known relation schemes. As these are the only tuples that the query is interested in finding, they are therefore the only ones that need to be free in the formula. Expressions are evaluated by substituting the various appearances of the free tuple variable \( x \) in \( f \) for tuples of the relation that \( x \) has as members; any set of substitutions that produces the Boolean result TRUE as a result is part of the interpretation of that formula (so a formula which contains only existential quantifiers is TRUE if a single substitution can be located, whilst a formula containing universal quantifiers requires all substitutions to be true in order to return TRUE).
For example:

\[ \{ x(\text{CAPITAL}) \mid \text{geog}(x) \land x(\text{CONTINENT}) = \text{"Europe"} \} \]

is an expression that asks 'what are the capitals of Europe?' by making \( x \) a tuple variable of \( \text{geog} \), and restricting the value of \( \text{CONTINENT} \) in \( x \) to "Europe" (note that \( x \) is free in the formula). All capitals of those tuples where \( \text{CONTINENT} \) is "Europe" would be TRUE and therefore satisfy the formula.

2.3.4 Conceptual Graphs and the Tuple Relational Calculus

Functionally, conceptual graph theory has many similarities to the relational model - it is also a value-oriented data model (although it possesses more varied structures than the relational model in order to increase the semantic richness of the domain being modelled), and most of its operations are also closed in a way similar to the operations of the relational model. However, it is a regrettable fact that, prior to this work, there has been very little work done in showing how conceptual graph operations may be expressed in terms of relational operations.

It can be seen that some conceptual graph operations have analogues at the relational level:

- graph projection returns a particular set of concepts from a graph - based on their typing and conceptual relationships - whilst the relational projection operation returns a particular set of domains from a relation scheme.

- graph join joins two graphs on a specified common concept, whilst the relational join operation joins two relation schemes on a specified common attribute name.

- graph restriction is more specialised than relational restriction, operating in tandem with projection - it restricts a particular concept or concepts and returns all projections in which those restrictions appear; relational select operates over a relation scheme, returning all tuples of a table in which one or more attribute names are restricted.
The appearance of such analogues should not really be surprising, considering the early use of the formalism as a database interface formalism. They are interesting in that they suggest that certain conceptual graph operations have a fairly close correspondence with relational operations but, unfortunately, this correspondence is incomplete; moreover, some conceptual graph operations have no obvious operational analogue in the relational model.

The aim of this section therefore, is to formalise and extend the above similarities to cover all the operations of conceptual graph theory.

In order to show how conceptual graph operations may be expressed in terms of relational operations, the first step must be to look at each operation of conceptual graph theory, and to see if they are affected in any way by the domain extensions described above. Examination of [Sowa 1976, 1984] reveals the following operations over simple conceptual graphs:

- the canonical formation operations of copy, restrict, join and simplify.
- the derived operations of projection and maximal join.
- the $\phi$ operator.
- the encapsulation operations of expansion and contraction.

In order to modify these conceptual graph operations for use with domain mappings (and thus a mapping to the relational model), it must be shown how the presence of domain mappings affect the functionality of these operations.

2.3.4.1 Canonical Formation Operations and Domain Mappings

The canonical formation operations of conceptual graph theory are not greatly affected by the presence of domain mappings, and they do not directly affect the function of the existing operations; however, some minor extensions to those operations must be observed in the presence of domain-extended conceptual graphs.
Definition 2.17 Use of Formation Rules in Domain-Extended Conceptual Graphs: if a formation rule takes one or more domain-extended conceptual graphs as its argument(s), then the following extensions to those operations need to be observed:

i. **copy**: as well as copying all concepts and relations of a domain-extended conceptual graph $u$, this operation should additionally copy any domain mappings present in $u$.

ii. **restrict**: this operation specialises referent and label fields; restriction of a type or relation label will modify the domain mappings of the participating concept (if a restriction of a type label) or concepts (if a restriction of a relation label). This issue is considered in more detail in the following section 2.3.4.3.

iii. **join**: this operation joins the concepts $c$ and $k$ in graphs $u$ and $v$; if $u$ and $v$ are domain-extended and of identical label, the concept that is resultant from the join possesses all of the domain mappings of both $c$ and $k$ - even if some of the domain mappings possessed by the resultant concept are identical.

iv. **simplify**: if two concepts $c$ and $k$ in a domain-extended conceptual graph $u$ have identical conceptual relationships $r$ and $s$ between them, then either $r$ or $s$ can be removed by simplification, so long as the domain mappings associated with that relationship are also removed from $c$ and $k$.

As shall be seen in section 2.3.4.2 (which deals with the encapsulation operators), changing a type or relation label requires recomputation of domain mappings for the affected concept(s).

2.3.4.2 Derived Operations

The derived operations of projection and maximal join are affected by the use of domain-extended conceptual graphs in a similar way to the formation rules from which they are derived.
Definition 2.18 *Projection into a Domain-Extended Conceptual Graph*: the operation of graph projection is not directly affected by the use of a domain-extended conceptual graph as either projector or projectee, but any projections of a graph $u$ into a domain-extended graph $v$ should possess the domain mappings of the subgraphs of $v$ into which $u$ projects. If $u$ is domain-extended, this makes no difference to the operation.

Definition 2.19 *Maximal Join of Domain-Extended Conceptual Graphs*: each concept join between $c$ and $k$ in the domain-extended conceptual graphs $u$ and $v$ being joined, must possess all of the domain mappings of both $c$ and $k$; if $c$ and $k$ have identical conceptual relationships $r$ and $s$ between them, then either $r$ or $s$ can be removed by simplification, so long as the domain mappings associated with that relationship are also removed from $c$ and $k$.

2.3.4.3 *Encapsulation Operations*

Because encapsulation operations change the shape of conceptual graphs by hiding or unhiding subgraphs, they may give rise to problems in a domain-extended conceptual graph $u$. Their major drawback is that their use changes the type of concepts in $u$ and, because domain mappings are allocated on the basis of typing information, performance of an encapsulation operation disturbs those mappings and requires recomputation.

Definition 2.20 *Expansion in a Domain-Extended Conceptual Graph*: expansion of a concept $c$ in a domain-extended conceptual graph $u$ replaces the label $l$ of $c$ with a more general label $m$. In addition to the additional domain mappings that $c$ possesses as a result of its join to the body of the definition of $l$, all other domain mappings of $c$ must be modified to take into account the new label $m$, as the domain mappings for the dyad:

$$[l] \rightarrow (R) \rightarrow [k]$$
will be different to the domain mappings of the dyad:

\[ [m] \rightarrow_{(R)} \rightarrow [k]. \]

where \( k \) is the label of another concept. A similar situation applies to the expansion of a relation - the domain mappings of the concepts \( c \) and \( k \) associated with the connecting relation \( r \) being expanded, are replaced by the domain mappings of those relationships in the relator of \( r \) that join to \( c \) and \( k \) as a result of the expansion of \( r \).

**Definition 2.21** Contraction in a Domain-Extended Conceptual Graph: contraction of a subgraph \( g \) of a domain-extended conceptual graph \( u \) into a concept \( c \) of \( u \) replaces the label \( m \) of \( c \) with a more specialised label \( l \); any domain mappings possessed by \( c \) that concerned \( g \) are lost, and the existing domain mappings of \( c \) must be modified to take into account the new label \( l \), as the domain mappings for the dyad:

\[ [m] \rightarrow_{(R)} \rightarrow [k]. \]

will be different to the domain mappings of the dyad:

\[ [l] \rightarrow_{(R)} \rightarrow [k]. \]

where \( k \) is the label of another concept. A similar situation applies to the contraction of a relation - if there exists a subgraph \( g \) of \( u \) and \( u = v \) (where there exists a relation definition \( r = \lambda x, y \ v \)), then the concepts \( c \) and \( k \) of \( g \) that correspond to the genera of \( r \) in \( g \) lose any domain mappings that concern \( g \), and gain domain mappings based on the dyad:

\[ [c] \rightarrow_{(r)} \rightarrow [k]. \]
Any change in the type of c or k as a result of contraction will lead to changes in all other domain mappings possessed by c or k.

The encapsulation operators assume a hierarchical partitioning of relation schemes, and this is not necessarily a good thing. To explain this, it is necessary to touch briefly upon a design consideration that is explained more fully in the following chapter. Consider the graphs:

\[
\text{[PERSON] \rightarrow (ATTR) \rightarrow [TALLNESS].} \quad \text{[MAN] \rightarrow (ATTR) \rightarrow [TALLNESS].}
\]

and, for purposes of exposition, assume that each of these graphs has domain mappings to separate relation schemes. If both encapsulation operators are operational, then the graph:

\[
\text{[PERSON: #1] \rightarrow (ATTR) \rightarrow [TALLNESS: *800].}
\]

would be stored in the relation scheme of the first graph. However, the graph:

\[
\text{[WOMAN: #2] \rightarrow (ATTR) \rightarrow [TALLNESS: *801].}
\]

has no relation scheme in which it can be stored - it cannot be stored in the first relation scheme because a loss of information would occur, and the type of the graph that maps to the second relation scheme is incompatible. There is nothing to stop the production of a third relation scheme to store this graph, but what if the graph:

\[
\text{[PRETTY-WOMAN: #3] \rightarrow (ATTR) \rightarrow [TALLNESS: *802].}
\]

is asserted? Another scheme is needed, and so it can be seen that the number of schemes needed is dependent on the complexity of the type and relation lattices. If these lattices are large, then this figure will be combinatorially explosive and create an unmanageable number of schemes over which any projection must range. Because such schemes are hierarchically related,

\[4\] The storage scheme outlined in Appendix E suffers from this problem to some extent - this is the primary appearance for its withdrawal from the Second International Conference on Conceptual Structures.
projection over such a storage system would require a large number of schemes to be unified
together in order to produce a complete answer - and the presence of the contraction operation
means that such a storage arrangement is necessary.

If the operation of contraction is not available (or rather, is used only in strictly controlled
circumstances), then a scenario where all graphs are stored in terms of their most primitive\(^5\)
types and relations can arise. In such a scenario, the number of relation schemes required for
storage would be less than or equal to the number of primitive relations in the relation lattice.

2.3.4.4 The \( \phi \) Operator

The \( \phi \) operator translates conceptual graphs into expressions in first-order predicate calculus. To
be useful with respect to the relational model, \( \phi \) must translate domain-extended conceptual
graphs into expressions in the tuple relational calculus. To show the correspondence between
conceptual graphs and the tuple relational calculus, an adaptation of the \( \phi \) operator that we shall
refer to as \( \phi_\Delta \) (\( \Delta \) refers to the database aspect) is now described.

**Definition 2.22** Modification of \( \phi \) in Order to Produce Formulas in Tuple
Relational Calculus: the \( \phi_\Delta \) operator translates simple conceptual graphs into legal
expressions in the tuple relational calculus. If \( u \) is any simple conceptual graph,
then \( \phi_\Delta u \) is an expression in the tuple relational calculus constructed by the
following:

i. if \( u \) is a conceptual graph with a single concept \( c \), assign a variable symbol \( x \) to it; the *free relation scheme* \( v \) has \( x \) as its single attribute name.

ii. if \( u \) is a conceptual graph with \( k > 1 \) concepts, assign a distinct variable
symbol \( x_1, x_2, \ldots, x_k \) to each concept; the *free relation scheme* \( v \) has \( x_1 x_2 \ldots x_k \) as its attribute names.

---

\(^5\) Primitive here means those types and relations that are direct subtypes and subrelations of the root nodes in each
lattice (although see the later discussion on metatypes in the type lattice for a minor modification to this meaning).
iii. $u$ can be fragmented into one or more connected subgraphs; each role domain mapping in a given subgraph belongs to the same relation scheme, and the overall set of domains mapped to in the subgraph contains no duplicates (the sole exception to this is if a concept is linking two dyads that both map to the same relation scheme - in such a case, it will possess two identical mappings and still be allowable as part of the subgraph).

iv. For each subgraph $g$ of $u$, create a tuple variable $t_1, t_2, ..., t_n$ and form a relation existential quantifier $\exists t(R)$ from an existential quantifier and a tuple atom $t(R)$, where $1 \leq k \leq n$ (the number of subgraphs in $u$) and $R$ is the relation scheme associated with that subgraph. Additionally, create a range atom, $R(t)$ for each subgraph $g$ in $u$, where $t$ is the tuple variable associated with $g$.

v. For each concept $c$ of $u$, create a view domain equality between the variable symbol of $c$ and a role domain mapping possessed by $c$. Each equality takes the form $v(x) = r(a)$, where $v$ is the free relation scheme, $x$ is the variable symbol of $c$, $r$ is the tuple variable associated with a subgraph in which $c$ appears, and $a$ is the attribute name mapped to by $c$ in that subgraph. If $u$ is simply $c$, perform the same task for the single conformity domain mapping associated with $c$.

vi. For each concept $c$ of $u$ appearing in multiple subgraphs, create a nonview domain equality of the form $t_a(a_1) = t_b(a_2)$, where $t_a$ and $t_b$ are the tuple variables of the subgraphs in which $c$ appears, and $a_1$ and $a_2$ are the attribute names associated with those concepts in the subgraphs $t_a$ and $t_b$ respectively.

vii. If a concept $c$ in $u$ is not generic, then for any subgraph $g$ of $u$ in which $c$ appears, create a referent equality of the form $t(R) = \text{referent}(c)$, where $t(R)$ is the tuple atom associated with the subgraph $g$.

$\phi_A u$ is constructed from the above elements in the following way:

$\{frs \mid req \land vde \land nde \land re\}$
where \(frs\) is the free relation scheme, \(req\) the list of relation existential quantifiers, \(re\) the set of range atoms, \(vde\) is the set of view domain equalities, \(nde\) the set of non-view domain equalities, and \(re\) the set of referent equalities. The internal elements of the last four are linked together by conjunctions, as well as each set being linked in the same way. Every part of the expression but the free relation scheme is referred to as the body of the expression.

\(\phi_d\) produces a legal tuple relational calculus expression that can be evaluated by turning it into relational algebra operations. The free relation scheme is effectively a view constructed from other schemes that satisfies the formula, and which can be used to extract data from secondary storage that can be used to answer a conceptual graph query. The tuples that satisfy the formula constructed by \(\phi_d\) are the interpretation of that formula, which is equivalent to the denotation of the graph from which the formula was constructed.

2.3.4.5 Demonstration of \(\phi_d\) and its Importance

To demonstrate the operation of \(\phi_d\) over a conceptual graph, look again at the graph used in Figure 2.2. This conceptual graph \(u\) has more than one concept, and so the free relation scheme for \(u\) is the tuple variable \(v(P_1, J_2, JT_3, D_4, PN_5, NIN_6, DN_7)\) with \(P_1\) mapping to [PERSON], \(J_2\) mapping to [JOB], and so on. \(u\) can be fragmented into four different relation scheme subgraphs \(R_1, R_2, R_3\) and \(R_4\); for each of these subgraphs, create tuple variables \(r_1, r_2, r_3\) and \(r_4\), as well as the relation existential quantifiers \(\exists r_1(R_1), \exists r_2(R_2), \exists r_3(R_3)\) and \(\exists r_4(R_4)\), and the range atoms \(R_1(r_1), R_2(r_2), R_3(r_3)\) and \(R_4(r_4)\). For the concepts in \(u\), create the following view domain equalities:

\[
\begin{align*}
  v(P_1) &= r_1(C1) \\
  v(J_2) &= r_2(C2) \\
  v(JT_3) &= r_3(C2) \\
  v(D_4) &= r_3(C2) \\
  v(PN_5) &= r_3(C3) \\
  v(NIN_6) &= r_3(C4) \\
  v(DN_7) &= r_4(C2)
\end{align*}
\]
Of the concepts in u, three appear in multiple subgraphs: [PERSON], [JOB] and
[DEPARTMENT]. For each of these, form the following nonview domain equalities:

\[ r_I(C1) = r_J(C1) \]
\[ r_I(C2) = r_J(C2) \]
\[ r_J(C2) = r_I(C1) \]

As the graph contains no referents, there are no referent equalities. All of these components can
be combined into the tuple relational calculus expression:

\[
\{ \forall (P_1 J_2 JT_3 D_4 PN_5 NIN_6 DN_7) \mid \exists r_I(R_1) \exists r_2(R_2) \exists r_J(R_3) \exists r_d(R_d)
\]
\[
(R_I(r_I) \land R_2(r_2) \land R_3(r_3) \land R_4(r_4) \land v(P_1) = r_2(C1) \land v(J_2) = r_3(C2) = r_d(C1))
\]

It should be noted that there is a variation between the expressions produced from domain-
extended conceptual graphs by \( \phi_a \), and the usual expressions of the tuple relational algebra. In
an earlier description of what makes up such expressions, it was stated that all the attributes R of
\( x(R) \) were a subset of \( U \), the universe of attributes. However, it is immediately noticeable that:

- the attributes R of the free relation scheme \( \nu(R) \) are brought into existence solely for the
  purposes of the above expression, and
- a large part of the body of the above expression is taken up with mapping those free
  relation attributes to other attributes in bound variables.

This is at variance with what is expected in a legal expression, and the reason for this variance is
solely due to the probable duplication of attribute names. For example, consider a query graph
in which the user wants to know which people own two books. As this graph will contain two
identical dyads that map to the same relation scheme, the view resulting from this query will
possess two attributes called BOOK, and such a relation scheme is not well-formed. This
problem can be addressed by application of the unique role assumption [Chang and Sciore
of the universal relation hypothesis, in which every attribute name must be unique in the universal relation scheme: although a universal relation exists only at the conceptual level, this assumption must hold at that level if the user is to be able to differentiate between the NAME of a person and the NAME of a ship in two of the relation schemes that contribute to the universal relation.

Based on application of this assumption then, the usual approach to this problem when forming Cartesian Products is to rename the duplicate attribute names in order to achieve product compatibility. As the difference between $\phi_{\Delta}$ expressions and standard tuple relational calculus expressions is essentially one of product compatibility, it shall be assumed that any attributes of $\nu(R)$ are admissible elements of $U$ (and so expressions produced by $\phi_{\Delta}$ are, by virtue of this variation, well-formed).

Although the extension of all conceptual graph operations is undoubtedly important if conceptual graph theory is to be used in the development of a hybrid system, the development of the $\phi_{\Delta}$ operator is certainly the most important extension. Such an operator would allow inquiries to be formed from a domain-extended conceptual graph; a tuple relational calculus expression could be generated from a projection into a domain-extended conceptual graph, and so could be used to extract all tuples from the relation schemes used in the expression - the expression would effectively become the 'retrieval arm' of the projection. Similarly, this operator would allow an instantiated domain-extended conceptual graph to be 'fragmented' over the relation schemes that it maps to, allowing the storage of the data in that graph and the retention of nothing more than a data-free graph and its mappings, that would allow the retrieval of that data via the route just outlined.

A similar approach is taken by $\lambda$-expressions, in which free variables are indicated by $\lambda$-variables instead of existentially quantified variables. Both of these approaches can be standardised under $\phi_{\Delta}$.
A ‘markered’ query graph can be represented as a λ-abstraction - as shown in Figure 2.7, which also shows an equivalent form in predicate calculus. All of these representations can be represented in the tuple relational calculus as the expression:

\{v(MONEY) \mid donations(v) \land v(YEAR) = 1981 \land v(PERSON) = \text{“Archer”}\}

in which the query marker and the λ-variable both define the attributes of the free variable v. If all variables in a λ-abstraction were λ-variables, the expression produced would be:

\{v(DONATE, PERSON, MONEY, YEAR) \mid donations(v) \land v(YEAR) = 1981 \land v(PERSON) = \text{“Archer”}\}

which is derived from the graph of Figure 2.8 overpage.

As a graph seemingly without query markers or λ-variables can produce the same expression as graphs without those structures, might it be argued that such structures are merely syntactic conventions? This isn’t quite the case, as they do have some semantic content; it is just that the ‘query’ graph above contains those semantics implicitly.
Figure 2.8 *A Query Graph without ? or λ-Variables*

The graph of Figure 2.8 is, in fact, a λ-abstraction in which the convention that an uninstantiated concept is implicitly a λ-variable (as [Sowa 1984] has observed) has been adopted. This implicit assumption about abstraction concepts means that an explicit list of λ-variables is no longer needed.

2.3.4.6 *Deficiencies of ϕ₄*

As the application of conceptual graphs to relational database retrieval has not been particularly well-developed, it is true to say that ϕ₄ is the most sophisticated system of linking conceptual graphs to relational database retrieval currently extant; its development is far more tractable than the vague pointers given by others in this area, and fulfils the need for a formal analysis of the subject.

However, it is still not complete as it only maps a subset of conceptual graphs to a subset of expressions in the tuple relational calculus, and the purpose of this section is to highlight those areas in which there is, as yet, no correspondence. For simple conceptual graphs, their domain-extended form will always translate into legal expressions in the tuple relational calculus via ϕ₄. To show an equivalence in expressive power though, it must also be demonstrated that all formulas of the tuple relational calculus can be mapped into conceptual graphs. To this end, it is not necessary to show a mapping between a full expression of the tuple relational calculus and a conceptual graph - it will be sufficient to show a mapping between a legal formula and conceptual graphs.
Assume, as when describing $\phi_\Delta$, that a series of domain mappings exists between a conceptual graph $u$ and attributes in some relation schemes. As the conversion conceptual graph structures into expressions in the tuple relational calculus was possible, these mappings can be used to take the constructors of the tuple relational calculus and convert them into conceptual graph structures. However, a number of these translations require new syntactic constructions if they are to succeed.

Given a conceptual graph $u$ and a legal formula $f$ of the tuple relational calculus, these structures can be defined:

Definition 2.23 Concept Comparison: the atomic formula $x(A) \theta c$ corresponds to a concept comparison $[A \theta c]$. For example, the formula:

$$x(\text{QUANTITY}) \leq 20$$

corresponds to the graph:

$$[\text{QUANTITY} \leq 20].$$

The concept comparison is a construction that compares the concept's referent to a constant $c$. Although this construction may look odd, it is preferable to the more obvious construction of Fig 2.9 because in the form $[A \theta c]$, the concept $A$ maps to a single set of role domain mappings; conversely, the form $[A] \rightarrow (\theta) \rightarrow [c]$ involves two concepts and two sets of mappings (even if one is a constant). As only a single attribute is being restricted, the second form would produce a questionable expression in the tuple relational calculus.
As an aside, it should be noted that the construct:

\[ \text{[NUMBER} > 0] \]

appears in [Sowa 1984] as a contracted form of the graph:

\[ \lambda x \text{[NUMBER}: *x]\rightarrow(\rangle\rightarrow\text{[NUMBER}: 0] \]

which is a \( \lambda \)-abstraction. This should be rejected for the reasons given on the preceding page.

**Definition 2.24** *Comparison Dyad:* the atomic formula \( x(A) \theta y(B) \) (where \( x \) and \( y \) may be the same tuple variable) has no analogue in conceptual graph theory, and corresponds to a *comparison dyad* \( [A]\rightarrow(\theta)\rightarrow[B] \). For example, the formula:

\[ stockinfo(x) \land x(\text{QUANTITY}) \leq x(\text{REORDER}) \]

where *stockinfo* contains the attributes *PARTNO*, *QUANTITY* and *REORDER*, corresponds to Figure 2.10.

It should be noted that a 'comparison dyad' formula in the tuple relational calculus doesn't always correspond to a comparison dyad as just described.

\[ [\text{PARTNO}] - \]
\[ (\text{AMOUNT})\rightarrow[\text{QUANTITY}] - \]
\[ (\leq)\rightarrow[\text{REORDER}: *x], \]
\[ (\text{AMOUNT})\rightarrow[\text{REORDER}: *x]. \]

**Figure 2.10 Conceptual Graph Containing a Comparison Dyad**
For example, the formula:

\[ \text{personnel}(x) \land \text{jobinfo}(y) \land x(\text{PERSON}) = y(\text{PERSON}) \]

where jobinfo contains the attributes PERSON and JOB, corresponds to Figure 2.11, which contains no comparison dyad as described in Definition 2.24. The equality between \(x(\text{PERSON})\) and \(y(\text{PERSON})\) - the same attribute in different schemes - instead indicates a join between the subgraphs represented by the two variables.

The introduction of \(\theta\)-comparisons into conceptual graphs means that some method of checking that the concepts participating in those \(\theta\)-comparisons are actually \(\theta\)-comparable must introduced; after all, constructions such as those in Figure 2.12 are nonsensical and obviously not \(\theta\)-comparable.

In order to guarantee \(\theta\)-comparability, the idea of metatyping is now introduced: if two concepts (and thus, their associated relational attributes) do not have the same metatype, they are not \(\theta\)-comparable and so the query graph in which they appear is badly-formed; similarly, if a constant within a concept comparison does not have the same metatype as the constant label, then it is not \(\theta\)-comparable and so the query graph in which it appears is badly-formed. Initially, three metatypes are proposed: numeric, string, and individual (a specialised sort of string).
Definition 2.25 Connectives in Conceptual Graphs and Tuple Relational Calculus Formulas:

- the formula \( \neg f \) corresponds to the query graph \(( f )\).
- the formulas \( f \land g \) and \( f \lor g \) correspond to the graphs:

\[
\begin{align*}
&f \quad g \\
&\text{and:}
\end{align*}
\]

\[
\begin{align*}
&( ( f ) \ ( g ) )
\end{align*}
\]

respectively\(^6\).

- the formula \( \exists x(R)f \) corresponds to a graph or subgraph \( f \) in which all variables of the constituent concepts are existentially quantified. This is implicit through \( \phi \).
- the formula \( \forall x(R)f \) corresponds to the graph:

\[
\begin{align*}
&( x \ ( f ) )
\end{align*}
\]

where \( f \) is a graph or subgraph in which \( x \) is free. As conceptual graphs do not support the universal quantifier \( \forall \), \( \neg \exists x(R)-f \), which translates into the above graph, is used.

---

\(^6\)This style of representing contexts is due to [Heaton 1994], and is preferable to the propositional form with an attached monadic (NEG) relation.
Negation in formulas of the tuple relational calculus implicitly corresponds to the set difference operator of the relational algebra. For example, in the formula:

$$geog(x) \land \neg \alpha(\text{CONTINENT}) = \text{“Europe”}$$

$x$ is all those tuples of $geog$ in which CONTINENT is not equal to Europe. Negation in conceptual graphs, on the other hand, is explicit - graphs such as that shown in Figure 2.13 assert that the child relationship between the two persons is false. Because the tuple relational calculus is based on a closed world assumption in which all data stored is true, negation can be anything that is not stored; conversely, conceptual graphs are based on an open world assumption, whereby anything that is not explicitly asserted as true or false is unknown.

Although conceptual graphs can operate on a closed world basis, it is preferable if they can operate on an open world basis. Therefore, $\phi_\Delta$ also needs to be extended to handle negation in an open world incarnation - replacing ‘negation as set difference’ with data stored in ‘negative’ relation schemes which are known to store explicitly-asserted negative data.

2.3.4.7 Revision of $\phi_\Delta$

Given these issues - $\theta$-comparison structures, metatyping, and ‘explicit’ negation - it can be seen that the original definition of $\phi_\Delta$ - which is here dubbed $\phi_{\Delta\alpha}$ - is incomplete. $\phi_{\Delta\alpha}$ therefore needs to be revised to take account of these issues, and this extended operator is dubbed $\phi_{\Delta\beta}$. 
Definition 2.26  Additional Constructors of $\phi_{\Delta \theta}$ over $\phi_{\Delta \alpha}$: the following constructors can be added to $\phi_{\Delta \alpha}$ in order to handle the issues of $\theta$-comparison structures, metatyping, and 'explicit' negation:

i  if a concept $c$ of a graph $u$ is a concept comparison, create a $\theta$-comparison $x(d) \theta k$, where $k$ is the constant in $c$, $\theta$ is the comparison operator in $c$, $x$ is the tuple variable corresponding to any subgraph of $u$ in which $c$ participates, and $d$ is the attribute name to which $c$ maps in $x$. The label of $c$ and $k$ must have the same metatype and thus be $\theta$-comparable.

ii if a graph contains a comparison dyad $[A] \to (\theta) \to [B]$ where $A$ is an attribute of tuple variable $a$ and $B$ is an attribute of tuple variable $b$ (which may be the same variable), create a $\theta$-comparison $a(A) \theta b(B)$. The concepts $A$ and $B$ must have the same metatype and thus be $\theta$-comparable.

iii if a graph $u$ is in an oddly-enclosed context, then a relation scheme $R_n$ exists that maps to $u$; any relation scheme $R$ is actually two schemes with the same attributes - $R_p$, which stores explicitly true data and is the scheme that has been previously referred in earlier examples as $R$, and $R_n$ - the 'shadow' relation scheme of $R_p$, which stores explicitly negated facts. Therefore, an oddly-enclosed $u$ maps to $R_n$ instead of $R_p$ in construction:

iv any lines of identity between contexts in a complex graph $u$ are nonview domain expressions, and represent equality between two attributes in different relation schemes.

The rule for constructing expressions must also be extended to take in $\theta$-comparisons $tc$:

$$\{frs \mid req \ra \land vde \land nde \land re \land tc\}$$
2.4 Correspondence Between Relational Constraints and Graph Constraints

The correspondence between conceptual graph theory and the relational model in both structural and operational terms has now been covered, and it has been shown how they must be extended both semantically and operationally in order to make that correspondence. The correspondence between both structures and operations makes this extended conceptual graph theory more highly developed than most semantic data models; and, as two of the components generally regarded as necessary in a data model [Date 1990] have been covered, it only remains to consider how conceptual graph theory corresponds to the third and final component - the constraint mechanisms of the relational model.

2.4.1 Constraints in the Relational Model

A constraint on data is used to maintain the integrity of the database, by ensuring that impossible configurations of values do not occur. As [Date 1990] has noted, most databases are subject to a large number of constraints, such as:

- supplier status values must be in the range 1-100.
- part weights must be greater than zero.

and so on. Most constraints - including the above - are domain specific, and the relational model possesses only two (perhaps three) general constraints that apply to all domains represented in a relational database - entity integrity, referential integrity, and perhaps conformity integrity.

**Definition 2.27 Entity Integrity:** no component of the primary key of a base relation is allowed to be null.

Null values can be anything, and so violation of this constraint would lead to potential violation of the uniqueness of primary keys.
Definition 2.28 Referential Integrity: the database must not contain any unmatched foreign key values, where a foreign key is an attribute of some relation $R$ whose values are required to match the primary key of some relation $S$. For example, a personnel relation would not be able to include details of a person based in the London office, if London did not exist in a sites relation that is also in the database.

The reasoning behind this constraint is also evident - if an entity $e_1$ in one relation refers to an entity $e_2$ in the same or another relation, then $e_2$ must exist.

Definition 2.29 Conformity Integrity: [Date 1990] describes a third constraint (which is here dubbed conformity integrity), which states that all values of a given attribute must be drawn from the relevant domain.

Some postrelational systems based on the relational model - most notably RMT [Codd 1979] - have an extended set of general constraints in order to deal with the more specialised structures used by that model.

2.4.1.1 Structure-Based Constraints

The relational model is also constrained by its structures - which, in turn, are shaped by functional and multivalued dependencies. Dependency can be viewed as a constraining property of the relations between two type domains and, as such, it is a valuable extra property that can be used to prevent uncanonical assertions; in a system based on relational storage, it is also one of the most useful pieces of information used to efficiently design the relational schemes that store data.

Informally, functional dependence is a property that operates between a number of data domains, and which indicates that data elements in a given domain or domains are constrained in what their values might be by the data elements in a domain or domains upon which they are functionally dependent.
Definition 2.30 *Functional Dependency*: \( X \rightarrow Y \) - or "domain \( X \) functionally determines domain \( Y \)" - means that for any value \( y \) such that \( y \in Y \), then there can only ever be one tuple which contains \( y \) in that domain, and there can only ever be a single \( x \) such that \( x \in X \) that can map to it.

This is very similar to one-to-one (1-1) cardinality. For example, it might be said that a person's name, address, and such like are functionally dependent on a national insurance number, because there is a 1-1 cardinality between the number and those items of data.

Definition 2.31 *Multivalued Dependency*: \( X \rightarrow \{ Y \} \) - or "domain \( X \) multi values domain \( Y \)" - is much the same as functional dependency, except that each \( x \) such that \( x \in X \) can map to a multiple \( \{ y \} \) such that \( \{ y \} \in Y \).

This is very similar to the one-to-many (1-N) cardinality. For example, a course may be taught in many rooms and at many times in a timetable, but only one course can be taught in a given room at a given time - which is a multivalued dependency between course and a room/time pair which acts as the key.

2.4.2 Constraints in Conceptual Graph Theory

How do the constraint mechanisms of conceptual graph theory correspond to the constraint mechanisms of the relational model? A salient fact that must be mentioned immediately concerns the status of keys - both primary and foreign - which simply have no real analogue in conceptual graph theory.

What is a key? Formally, a key \( K \) of a relation \( R \) in the relational model is unique if no two tuples of \( R \) have the same value for \( K \); \( K \) is also minimal if no components of \( K \) can be removed without destroying the uniqueness property. A set of attributes that is both unique and minimal for \( R \) can be a key for \( R \). Dependency information can certainly indicate certain attributes of a relation scheme as being unique - for example, a scheme based on the graph:
[PERSON]→(ATTR)→[NI-NUMBER].

in which the relationship between the two concepts has a one-to-one cardinality and mandatory membership of the relationship on both concepts. Any resultant scheme mapped to by this graph would be unique on both PERSON and NI-NUMBER attributes, so cardinality and membership information can allocate the property of uniqueness to single attributes or groups of attributes. As such groups are also minimal, any relation scheme may contain multiple candidate keys - groups of attributes that satisfy the uniqueness and minimality requirements. There is no way for a conceptual graph to identify which key is the primary key from these candidates.

Given this, what relationship does domain-extended conceptual graph theory have with relational constraints? As regards entity integrity, the fact that completely general generic markers (such as nulls) are not allowed in the extended theory detailed in this chapter guarantees the entity integrity constraint - as no component of a tuple (and so no component of any key) will be null.

Referential integrity is slightly more involved, and this is the only real area where the lack of a primary key makes much difference. The property of referential integrity means that the database must not contain any unmatched foreign key values. Foreign keys are generally used in 'relationship' schemes - these are relations that normally consist solely of two attributes, both of which are keys of 'entity' schemes.

To illustrate this: assume the existence of a supplier scheme and a parts scheme - both of which refer to 'concrete' entities; to record which parts are shipped by which suppliers, a 'relationship' scheme consisting of the primary keys of both the supplier and parts entities could be formed. Such an arrangement is illustrated in Figure 2.14. Now consider if the entity-relationship diagram of Figure 2.14 became this graph:

[SUPPLIER]→(S_P_RELATIONSHIP)→[PART].
with similar relations. As each concept is an entity in itself, there is no need for explicit 'entity' schemes, which leaves just the 'relationship' scheme. Because the attributes of S_P_REL no longer have to refer to other schemes, they are no longer foreign keys and so the referential integrity constraint becomes redundant. All that needs to be checked is that the marker of each concept has the right conformity for its type, and this is virtually identical to the conformity integrity constraint (Definition 2.29) - which, in its turn, is virtually identical to the conformity relationship "��" of conceptual graph theory.

The correspondence with relational constraints aside, conceptual graph theory does not really contain any general constraint mechanisms; perhaps the closest thing it possesses to a general mechanism is canonicity7.

**Definition 2.32 Canonical Graph** [Sowa 1984]: a conceptual graph is normally canonised by any of the following processes:

- *formation rules*: a canonical graph may be derived from another by application of a formation rule.

---

7 Domain-specific constraints are expressible in the existential graph logic of Peirce.
• *insight*: a graph may be arbitrarily be assumed to be canonical.

The primary use of canonical graphs is to ensure *plausibility* of graphs - some graphs are asserted as canonical by insight, and any other graphs must be derivable from that *canonical basis* by application of formation rules. However, [Sowa 1984] defines a *canonical model* as:

- a type lattice $T$.
- a set of individual markers $I$.
- a conformity relationship to map types in $T$ to markers in $I$.
- a canonical basis $B$, in which all types are in $T$ and all markers are in $I$.

This requires some extension. The most obvious extension is the addition of a relation lattice $R$, as the formation rule of restriction should operate over such a structure. $B$ can also operate as a canonical basis without explicit reference to $I$, as any instantiated concept can be generated by the application of restriction to an uninstantiated concept.

As defined above, $B$ acts as an ‘upper bound’ beneath which any graph is canonical. However, situations may arise where the application of canonical formation rules may give rise to badly-formed graphs. For example, the graph:

$$[\text{MAN}] \rightarrow (\text{SPOUSE}) \rightarrow [\text{WOMAN}].$$

may be a graph in $B$; given the right circumstances in $T$ though, it may be possible to generate the graph:

$$[\text{CATHOLIC_CLERGYMAN}] \rightarrow (\text{SPOUSE}) \rightarrow [\text{WOMAN}].$$

In the absence of domain-specific constraints, $B$ may be extended to a true basis set of graphs $t$ and a false basis set of graphs $f$. Within $t$, all graphs can be joined maximally as there are no contextual boundaries to prevent this whilst within $f$, no nesting of contexts is permitted.
Definition 2.33 Canonicity over an Extended Basis: a conceptual graph $u$ is canonical if, given a true basis $t$ and a false basis $f$, it is both a specialisation of $t$ and a generalisation of $f$. If $u$ is a generalisation of $t$ or a specialisation of $f$, then it is uncanonical.

The false basis therefore acts as a 'lower bound' on conceptual graphs in an application domain. Canonicity is a mixture of general constraint and domain-specific constraint - it can generally prevent two conceptual graphs joining together if the result is uncanonical, but this requires domain-specific information in order to operate. A canonical model would be able to prevent the example of incompatible domain join seen in the Introduction, where a name of a ship could be joined to the name of a person; such a join would be uncanonical.

2.4.2.1 Dependency and Peirce Logic

The mappings of $\mu_R$ are between concepts in conceptual graphs and attributes in relation schemes, and it goes without saying that those relation schemes should be normalised. As conceptual graphs are being mapped onto the relational model, it makes sense to use a tried and tested mechanism - normalisation - to construct the relation schemes to which $\mu_R$ map to.

If an application domain is static, a set of normalised schemes - and $\mu_R$ - can be calculated by the domain designer manually, and they will never change; however, most application domains are not static, and changing circumstances within those domains may lead to circumstances where the initially normalised relation schemes are no longer capable of modelling those domains. Thus, the creation and maintenance of $\mu_R$ becomes a subsidiary problem of maintaining normalisation in a dynamic application domain.

The ability of the theory to change $\mu_R$ as the application domain changes - without user intervention - is of key importance, because it means that the user will not have to tinker about with relational storage if the application domain changes. This ability is part of the general problem of schema evolution over time, which has been recognised for some time within the
relational model\textsuperscript{8}, and there has been much research into areas surrounding the derivation of relation schemes from various sources that are of use in the pursuit of this aim. [Bernstein 1976] is perhaps the earliest source to give an algorithm for the extraction of schema structures from functional and multivalued dependencies, whilst [Navathe 1980] has dealt with the translation of one model's storage into another. [Flynn and Laender 1985] maps a NIAM diagram into a more internal representation and [Leung and Nijssen 1987, 1988] have dealt with the derivation of a design from the NIAM design formalism. Related to this are [Staley and Anderson 1985; Briand et al 1985], which both offer schemes for converting entity-relationship diagrams into relation schemes. [Mannila and Räihä 1987] have shown how dependencies can be extracted from existing relational database schemes and the data they contain, but such an approach is rather close to the idea of machine learning and is outside the scope of this work. Moving closer to the notion of automatic adaptation of storage, [Sockut and Goldberg 1979] operates schema evolution over network and hierarchical databases, changing structures at the physical level when there are changes at the logical level; [Williams et al 1986] considers the intensity of table use as a yardstick for scheme reorganisation, and, similarly, [Markowitz and Makowsky 1987] have shown how relational database schemes can be adaptively reorganised through the use of a modified entity-relationship diagram (although the fact that it is based on the entity-relationship formalism does rather limit it as a useful tool for deductive database design. Like the above approach, it attempts to perform incremental schema evolution through a diagrammatical interface, and it is relationally based).

2.4.2.2 $\mu_R$ as a Subsidiary Problem of Normalisation in a Dynamic Application Domain

What information is needed to maintain $\mu_R$ (and also normalisation) in relation schemes? The entity-relationship modelling approach has a number of similarities to conceptual graphs, and is used to design storage-efficient relational schemes; leaving aside the observation that the entity-relationship formalism is less powerful than conceptual graphs in what it can do, they differ from conceptual graphs in that they contain explicit declarations of two relational properties -

\textsuperscript{8} Although in other paradigms - particularly the object-oriented model - the issue of schema evolution seems to be of greater interest to researchers. For example, see [Hudson and King 1989], [Lerner and Habermann 1990], and [Tresch and Scholl 1993].
Cardinality and membership. Cardinality of a relation is simply a limitation on how many instances may take part in a relationship. Within the entity-relational approach, four cardinalities are possible:

- **one to one** (or 1-1): for example, a person may only possess a single taxcode at any given time, and that taxcode only applies to one person.
- **one to many** (or 1-N): for example, a man may have many children, but each child only has one father.
- **many to one** (or N-1): for example, a city's geography allows it a single latitude, but there can be many cities at that latitude.
- **many to many** (or N-M): this is basically the default, in that it places no restriction on the relationship.

Membership of a relation simply specifies whether all instances should participate in the relationship or not. Two memberships are possible:

- **mandatory** (or obligatory): every instance must participate in the relationship. For example, every national insurance number issued must belong to a person - it cannot exist in isolation.
- **optional**: every instance may participate in the relationship, although it isn't compulsory. For example, not every person has to have a national insurance number (as they are assigned to citizens of a certain age and above).

Possession of these two properties allows the construction of storage-efficient relational structures; however, regular conceptual graphs contain no explicit mechanism for the assertion of such information. Given the origins of conceptual graph theory in the area of relational databases [Sowa 1976], it is mildly surprising that it contains no explicit syntactic facility for asserting information about the cardinality and membership properties of conceptual relations. These properties are vitally important to the efficient construction of relational storage structures, and so various modifications to the syntax of conceptual theory have been proposed in an attempt to remedy this deficiency.
2.4.2.3 Explicit Representation of Cardinality and Membership in Conceptual Graphs

Perhaps the most advanced of these approaches is that of [Creasy and Ellis 1993], who favour an explicit syntactic extension to conceptual graph theory that modifies the arcs of conceptual relations to include cardinality information. Graphs carrying cardinality and membership information in this syntax bear an interesting resemblance to more primitive entity-relationship diagrams and their spinoffs⁹, although the advantage of using such extended conceptual graphs over traditional entity-relationship diagrams is obvious - all of the information that an entity-relationship diagram models can still be modelled, whilst other parts of conceptual graph theory can be used to impose additional restrictions and consistency of data; in short, such an extended syntax provides a unified method of modelling storage structures and the information that can appear in them.

Although such an explicit approach is completely tractable, it does rather go against the grain of one of the research's stated aims - that of shielding the user from needing to know anything about the underlying database. The desire to maximise the effect of this aim has therefore led to the investigation of non-explicit ways of representing cardinality and membership in conceptual graphs¹⁰. In pursuit of this goal, an implicit approach where cardinality and membership information is extracted from existing graph structures - instead of extending the formalism - is proposed. With such an approach, it becomes possible to automatically detect changes in the application domain that will lead to renormalisation and a change in $\mu_R$, as well as removing the need for the user to be aware of the database component in design.

---

⁹ Such as the NIAM data design formalism - [Campbell and Creasy 1992] show similarities between NIAM and conceptual graphs, and extensions to conceptual graph theory are proposed in order to provide direct equivalents to NIAM constructions.

¹⁰ It is worth emphasising that explicit approaches are not thought to be without merit as, even if the information in such extended conceptual graphs is not explicitly used to form database structures, it is still of value as an additional form of canonical restriction - for example, the fact that a person cannot possess more than one social security number can be used to reject inputs that try to assert such a situation. It is simply felt that a less explicit approach is more in line with the original aims of the research.
2.4.2.4 Extraction of Normalising Information from Peirce Logic Formulas

It can be shown that both cardinality and membership information can be extracted from conceptual graphs, and so additional syntactic constructions are not required. If the property of cardinality is considered, three of the four cases cited above can be represented as formulas in Peirce logic (assume the N-M case is default):

- **1-1**: consider the following pair of graphs in Figure 2.15(a). They can be read as 'it is false that a person has two social security numbers as attributes' and 'it is false that a social security number can be had as attribute by two different people'; this pair of graphs together state that a person cannot canonically have more than one social security number, and that a social security number cannot canonically be allocated to more than one person - the relationship:

  \[
  \text{[PERSON]} \rightarrow \text{(ATTR)} \rightarrow \text{[SS-NUMBER]}
  \]

  is therefore one-to-one in nature.

- **1-N**: consider the graph in Figure 2.15(b). This graph can be read as 'it is false that a person has two other persons as a father'. This doesn't prevent a person from being the father to more than one other person, and so the relationship:

  \[
  \text{[PERSON: *x]} \rightarrow \text{(FATHER)} \rightarrow \text{[PERSON: *y]}
  \]

  is therefore one-to-many in nature.

- **N-1**: consider the graph in Figure 2.15(c). This can be read as 'it is false that a book has two publishers'; this is more or less the reverse of the previous case in that it doesn't prevent a publisher from publishing more than one book, and so the relationship:

  \[
  \text{[BOOK]} \rightarrow \text{(PUBLISHER-OF)} \rightarrow \text{[PUBLISHER]}
  \]

  is one-to-many in nature.
Moving on to the property of membership, mandatory membership can also be represented as a Peirce formula. The essence of representing mandatory membership of a relationship is to make sure that each entity participates in it. Consider the graph in Figure 2.16.
This is simply an implication stating that 'if a person x exists, then that person x has a social security number'. Therefore, every person must have a social security number as attribute and so its participation is mandatory. It also just happens to be a Skolem function - wherever an existentially quantified variable depends upon a universally quantified variable, there is a function between them\textsuperscript{11}.

\textsuperscript{11} This Skolem dependency has also been noted by [Creasy and Ellis 1993]. It might also be noted that a many-to-many cardinality is a pair of Skolem functions pointing in opposite directions.
In the first graph of Figure 2.17, the Skolem function indicates a dependency between PERSON and SS-NUMBER - every person must have a social security number. In [Sowa 1990], Skolemisation is used to bind an directed actor to the two participating nodes, as in the second graph of Figure 2.17. However, actors have been abandoned in their dependency role because they are explicit, and this Skolem information is instead used to generate and maintain relation schemes implicitly.

2.4.2.5 Use of Normalisation Information in Relation Scheme Maintenance

Although cardinality and membership information can appear implicitly in a domain of discourse, this does not actually solve the problem - although they are the tools needed to perform any renormalisation and changes in $\mu_R$. The next step is to show how something useful can be done with this information beyond its naive use as a canonical restriction mechanism.

\[
\begin{align*}
[\text{BOOK}] - \\
&\text{(ATTR)} \rightarrow \text{[ISBN-NUMBER]} \\
&\text{(ATTR)} \rightarrow \text{[YEAR-PUBLISHED]}. \\
\end{align*}
\]

(a) Conceptual Graph

$\lambda_1 (\text{BOOK ISBN-NUMBER})$ $\quad$ $\lambda_2 (\text{BOOK YEAR-PUBLISHED})$

(b) Naive Relation Schemes

$\lambda_3 (\text{BOOK ISBN-NUMBER YEAR-PUBLISHED})$

(c) Posted Scheme

Figure 2.18 A Conceptual Graph and Posting
Looking at a conceptual graph composed of concepts and dyadic conceptual relationships, there are three possible ways in which two directed dyads can be connected together:

- \([A] \leftarrow \langle R \rangle \leftarrow [B] \rightarrow \langle R \rangle \rightarrow [C]\)
- \([A] \rightarrow \langle R \rangle \rightarrow [B] \leftarrow \langle R \rangle \leftarrow [C]\)
- \([A] \rightarrow \langle R \rangle \rightarrow [B] \rightarrow \langle R \rangle \rightarrow [C]\)

In the absence of any other information, it is assumed that each dyad maps to a binary relation scheme - so the first graph above would naively map to the schemes:

\[
\lambda_1 (B, A) \quad \lambda_2 (B, C)
\]

and so on. Depending on the cardinality and membership properties of the linking concept with regard to the dyads in which it participates\(^{12}\), the role domain mappings of the linking concept with respect to those two dyads may be combined, and the two relation schemes may also be combined as a result. For example, consider the case of the graph in Figure 2.18(a): if information could be extracted stating that each of these dyads were one-to-one cardinalities - and that all memberships were mandatory - then the naive schemes of Figure 2.18(b) would no longer be normalised, and the scheme of Figure 2.18(c) would be required instead. There are twelve such posting events where this merging of naive schemes can occur, and these are outlined in detail in Appendix A of this thesis.

2.5 Usefulness of Domain-Extended Conceptual Graph Theory

It has been shown how conceptual graph theory corresponds to the relational model in three areas - structures, manipulations, and constraints. Although it is often stated that conceptual graphs have more expressive power than relational manipulation languages, little formal reason

\(^{12}\) Note that this is not the same as a role domain mapping - role domain mappings are derived from this information.
has previously been given to back up such a claim. However, it is possible to compare the expressive power of two manipulation systems formally [Maier 1983].

**Definition 2.34 Comparison of Expressive Power of Two Systems:** a manipulation system $M_1$ is as expressive as a manipulation system $M_2$ if, for every legal expression $E_2$ of $M_2$ and every database scheme compatible with $E_2$, there is a legal expression $E_1$ of $M_1$ such that $E_1$ is equivalent to $E_2$.

$M_1$ and $M_2$ may well be equally expressive; for example, it can be demonstrated that the relational algebra and the relational calculus are equally expressive. The relational calculus is therefore relationally complete as well. It has been shown that the tuple relational calculus is equivalent to a subset of conceptual graphs that are domain-extended (Definition 2.10) and modified to deal with $\theta$-comparisons (Definitions 2.23 and 2.24) and explicit negation (Definition 2.25) - and, as the tuple relational calculus is as expressive as relational algebra, it is therefore relationally complete and employable as a relational definition and manipulation language.

Conceptual graph theory is also more expressive than the relational algebra and equivalent systems, because expressions exist in the extended conceptual graph theory that cannot be represented by the tuple relational calculus - such as hierarchical information, the use of canonical structures to guarantee consistency, and so on.

2.5.1 Domain-Extended Conceptual Graphs and Datalog

The best known example of a deductive database language is *Datalog* [Ullman 1988; Ceri et al 1989] - which is a language that is derived from the Prolog logic programming language, and which certainly has greater expressive power and semantic capture than the languages of the relational model.

In its purest form, Datalog is a pure Horn clause language without function symbols or negation. It can be considered as a variant of Prolog, but with a set-oriented semantics that is independent
of clause order. In its pure form, Datalog doesn't support things like disjunction (although [Ceri et al 1989] details research efforts into extensions in this area).

2.5.1.1 Semantics of Datalog

An interpretation of a Datalog program is the set of instantiations from all extensional predicates in the program (these are tuples from the extensional relation schemes corresponding to those predicates), as well as the set of instantiations that can be derived from the extensional tuples by the application of the program's rules; all tuples in this interpretation are true, and so it is also a model.

A pure Datalog program may be recursive but not have negated goals or subgoals; conversely, the languages of the relational model have negation but not recursion.

2.5.1.2 Negation in Datalog and Extended Conceptual Graph Theory

Each Datalog program without negation has a unique minimal model, which is a particular model in which the modification of any instantiation of a predicate from TRUE to FALSE would render the model inconsistent. However, the addition of negation can lead to inconsistency in Datalog programs unless certain precautions are taken. For example, the Datalog program:

\[ P1 (x) \leftarrow \neg P2 (x) \]
\[ P2 (x) \leftarrow P1 (x) \]

will not generate a minimal model.

To guarantee the production of a minimal model, stratification is required - this requires that a Datalog program with negated subgoals be organised into modules (or strata) that each have a minimal model. The key point of any stratifying strategy is that for any predicate \( x \), \( x \) cannot be defined with reference to a predicate \( \neg y \), where \( y \) itself is defined with reference to \( x \). More
simply, \( \neg x \) can only appear in the body of a rule if (and only if) it has already appeared in other rules that are independent of the rule in which \( \neg x \) appears. Avoidance of such situations separates a Datalog program into strata, each of which will have a unique least model. The above example is not stratified, as the goal predicate \( \text{P1} \) is defined in terms of the negated subgoal \( \text{P2} \), which (as a goal) is itself defined in terms of the subgoal \( \text{P1} \).

Is extended conceptual graph theory affected by such requirements? After all, Peirce logic explicitly uses negation in the construction of complex conceptual graphs.

Complex conceptual graphs are not restricted to Horn clause formulas, and so may represent disjunctive information quite comfortably. As regards the stratification problem faced by Datalog, a system based on Peirce logic does not suffer from such problems as a domain containing unstratified formulas would be inconsistent and would therefore be rejected. If the unstratified Datalog program is considered:

\[
\begin{align*}
\text{P1} (x) & \leftarrow \neg \text{P2} (x) \\
\text{P2} (x) & \leftarrow \text{P1} (x)
\end{align*}
\]

again, the first formula is equivalent to the conceptual graph:

\[
( ( \text{[P2: *x]} . ) ( \text{[P1: *x]} . ) )
\]

(which, in the absence of other information, is unknown and so its addition to a domain will not cause inconsistency), whilst the second formula is equivalent to the conceptual graph:

\[
( \text{[P1: *x]} . ( \text{[P2: *x]} . ) )
\]

Peirce theorem proof is able to detect that a domain containing both of these graphs may give rise to inconsistency and so rejects it. Any set of complex conceptual graphs that are consistent with respect to Peirce's \( \beta \)-rules will be stratified and so guarantee a unique minimal model.
2.5.1.3 Representational Clumsiness of Prolog/Datalog

Extended conceptual graph theory is a highly abstract approach with less need for implementation-specific knowledge than Prolog- and Datalog-based systems, and it can also do things that those systems cannot do. For example, the operation of modus tollens in Prolog requires a Prolog program looking something like:

Q :- P.
Q :- !, fail.  (the closest that can be got to 'not Q' - heads can't be negative)

If an attempt to prove P is made, a Prolog-based system will answer FALSE, because there is no rule with P as its head; if an attempt to prove Q is made, a Prolog-based system will answer FALSE, but for the wrong reasons. The conceptual graph basis of C-GRASS allows modus tollens to be modelled directly, because conceptual graphs have a cleaner, more general syntax:

( P ( Q ) )
( Q )

If an attempt to prove P is made here, it is FALSE because Q is false - classical modus tollens.

Prolog-based systems are also limited by their restricted Horn Clause syntax in other ways. For example, it is not possible to use the statement:

Q :- P.

as a query in Prolog. Conversely, the conceptual graph:

( P ( Q ) )

is perfectly capable of being processed as a query by Peirce logical mechanisms.
2.5.1.4 *Comparison of the Two Systems*

It is known that safe nonrecursive Datalog with negation is equal to the relational calculus, and so is relationally complete; with recursion, it formally has greater expressive power than the relational calculus. However, the representation of negation and disjunctive information in Datalog requires the user to stratify any program using negation, and this may not be a trivial problem in a complicated domain. Datalog is also a low-level system, with direct mappings from extensional predicates to relation schemes.

The extended conceptual graph theory described also possesses the properties of recursion, as well as the representation of negation and disjunctive information, and is also formally more expressive than the relational calculus. However, it also scores over Datalog in several areas:

- **explicit stratification in the presence of negation is not required, but is automatic.**
- **it is based on an open world approach, whereas Datalog is based on a closed world approach; this allows queries to evaluate to UNKNOWN, as well as TRUE and FALSE.**
- **modelling of disjunctive and other types of information is cleaner than in Datalog (even though Datalog is itself a cleaner version of Prolog).**
- **it operates at a higher level of abstraction than Datalog - the user need only communicate in terms of domain information, whereas Datalog still requires some specification of implementation-specific knowledge to organise extensional predicates.**
- **Prolog possesses nothing like a canonical mechanism, and has no implicit consistency mechanism.** For example, the program:

  Q :- P.
  Q :- !, fail.
  P.

is inconsistent (both Q and 'not Q' can be derived from it), yet is acceptable by Prolog.
2.5.2 Comparison of Domain-Extended Conceptual Graphs and Other Data Models

The extended form of conceptual graph theory described above is more complete than the vast majority of semantic data models; and whilst it may not have the computational power of the object-oriented database model, it possesses many structural ideas in common with it, whilst offering vastly superior semantic capture.

2.5.3 Conclusions

Many semantic modelling techniques exist that are little more than structure, but the contents of this chapter hopefully show that conceptual graphs are a proper data model. They possess:

- a *structure definitional* component (the graphs of conceptual graph theory are similar in many respects to the relations of the relational model, and conceptual graph theory has additional structures such as inheritance lattices, definitions, and so forth, which increase the semantic richness and expressiveness of any application domain represented).
- a *constraint* component (metatyping of concepts, canonicity of graphs, consistency of domain through Peirce proof mechanisms, etc).
- a *manipulative* component, as most of this chapter has been spent in showing, although outside of querying, conceptual graph theory also has other operations for operating over those structures - such as inheritance lattices - that play no direct part in querying.

As regards manipulation, the tradeoff between query power and query efficiency in the relational model is obviously a fine one, and the relational algebra gives a not insignificant amount of expressive power whilst still being very efficient. Any manipulation system operating over relational structures must therefore translate into the relational algebra in order to achieve maximum query optimisation, and shunt any of its parts that do not map to the relational algebra into processing areas where they will not affect this translation.

Some evidence has also been seen that the extended theory outlined in this chapter is preferable to Prolog and its related approaches, in that it possesses a more principled treatment of many
important semantic issues, whilst generally offering stronger consistency and a cleaner syntax to the user.

In short, domain-extended conceptual graph theory is a complete data definition and manipulation system that subsumes the relational model both structurally and operationally, whilst providing greater semantic capture and additional constraint mechanisms - but can it be successfully implemented?
CHAPTER 3
Design and Description of C-GRASS

C-GRASS - the Conceptual Graph Relational Adaptive Storage System - is a loose-coupled hybrid system based on the Oracle relational database management system and a conceptual graph interface layer written in the C language (see Figure 3.1). Although there has been much talk of conceptual graphs being extended into 'deductive object-oriented conceptual databases' [Wuwongse and Ghosh 1992], conceptual graphs can already be seen as a kind of object-oriented logic (possessing many object-oriented features such as hierarchies, encapsulation, and so on) with a strong deductive capability (Peirce logic), and C-GRASS is almost certainly the most completely realised version of this idea to date.

Figure 3.1 Architecture of C-GRASS
The interface layer of C-GRASS acts as its conceptual schema level, being a formal description of the application domain; and apart from where interaction with the logical schema is required - such as in the derivation of graph-relation scheme mappings and scheme construction - it is not concerned with description of what is stored (such as data elements, access paths, integrity constraints, and so on - and the representation of the application domain inside the computer).

C-GRASS has a system for the extraction of both mandatory membership and functional dependencies from the application domain, and this allows C-GRASS to be both declarative and adaptive: because the extraction of such information can be used to calculate the structure of underlying relation schemes, C-GRASS is capable of both creating and adapting those structures automatically, without any need for user intervention; the use of the $\mu_R$ mappings generated by those procedures also means that all interaction with the database management component is handled without the user requiring any implementation-specific knowledge of how the domain is stored; the $\phi_A$ operation and its adaptations translate domain-specific conceptual graphs into SQL instructions. This approach is more sophisticated than systems such as CSL-0 [Poesio 1987a, 1987b] that are based on binary relational storage - although it should be noted that very early versions of C-GRASS also used such a strategy (for example, see [Bowen and Kocura 1993] - included in this thesis as Appendix D).

C-GRASS is, in its primary incarnation, a research tool with the aim of investigating if conceptual graphs can solve the problems of the relational model. As its aim is primarily to increase the semantics of the relational model, and to increase the range of application domains that the relational model is good at modelling, it isn’t meant to be used over complex domains such as those which are better tackled by the object-oriented paradigm (with highly complex operations, for example).

3.1 Chapter Objectives

The purpose of this chapter is a description of the design of C-GRASS - how it is structurally organised, what operations it is capable of performing, and so on. Like the preceding theoretical
chapter, it is broken down into three main sections concerning conceptual graphs - dealing with the design of structures, operations, and constraints respectively. Before such graph matters can be discussed however, a necessary precursor is a discussion of storage matters, upon which many graph issues are dependent in C-GRASS.

3.2 Database Issues I: Storage Strategy of C-GRASS

Looking back at the objectives that were outlined in the Introduction, any storage strategy employed by C-GRASS has to satisfy the following requirements:

- it must use a commercial relational database system.
- it must operate with at least some degree of efficiency.
- it must require no user intervention either to create or modify the storage structures - ideally, the user should be unaware that a database package is being used to store domain information.

Additionally:

- it must be able to deal with information that is traditionally non-relational, such as hierarchies.

This section looks at two relational storage methods investigated in the attempts made to satisfy these requirements - N-ary and binary relational - and considers why these approaches are likely to find some of these requirements difficult to satisfy. It then proceeds to describe the strategy finally adopted, and the reasoning behind the choice made.

3.2.1 Storage Strategies and Storage Requirements

Traditional relational database storage strategy can be characterised as N-ary Relational - using multiple relation schemes of variable arity. However, simpler approaches based on relation
schemes of limited arity are also popular; the most well-known of these is the *Binary Relational* approach.

Bearing in mind the requirements detailed above, the more 'traditional' N-ary relational approach might seem rather unattractive - after all, the problems of achieving an efficient and irredundant scheme design through normalisation are well known. In addition, the binary relational approach has a definite advantage over the traditional approach in this area, in that any file design required is minimal or even non-existent.

Early versions of C-GRASS employed an binary relational approach to storage; however, this was later rejected in favour of an N-ary relational approach.

### 3.2.2 The N-ary Relational Approach

The N-ary relational approach can be characterised as the 'traditional' view of relational data storage - in that techniques such as normalisation are used to avoid data redundancy and null values wherever possible. Given that this approach is the norm when implementing application domains in the relational model, it thus needs little explanation here - although [Maier 1983] and [Date 1990] are recommended to the reader. It is compositional in nature as, subject to normalisation constraints, the arity of relation schemes is maximised wherever possible.

#### 3.2.2.1 Advantages of N-ary Relational Approach

The advantage of the N-ary relational approach is simple - the larger the relation scheme, the more concentrated and connected is the data that can be stored within it; and this means that the number of relational joins required to answer a query should be lessened - a desirable property, considering the expense of relational join operations [Mishra and Eich 1992] - and this makes for efficient retrieval. For example, it is more efficient to retrieve the national insurance number, surname, date of birth and address of all those people with a certain tax code from the normalised TAX relation scheme:

\[
\text{TAX (NAT-INS-NO SURNAME DATE-OF-BIRTH ADDRESS TAXCODE)}
\]
than it is from the (possibly) unnormalised relation schemes:

- **HAS_SURNAME**: (NAT-INS-NQ SURNAME)
- **HAS_BIRTHDATE**: (NAT-INS-NQ DATE-OF-BIRTH)
- **HAS_ADDRESS**: (NAT-INS-NQ ADDRESS)
- **HAS_TAXCODE**: (NAT-INS-NQ TAXCODE)

### 3.2.2.2 Disadvantages of N-ary Relational Approach

The only major drawback with this approach is that it tends to require a considerable amount of pre-implementational work (generally normalisation) in order to ensure data irredundancy. The information represented in them is also 'flat' and highly regimented, whereas conceptual graphs are heterogeneous in terms of flatness (they generally have some degree of hierarchical information) and structure (a graph may be composed of an arbitrary number of arcs and concepts).

### 3.2.3 The Binary Relational Approach

In direct contrast to the 'compositional' approach outlined above, systems based on the binary relational approach [Frost 1986; Copeland and Khoshafian 1985] are 'decompositional' - essentially representing the application domain in terms of binary relationships between atomic entities.

Binary relational systems generally tend to employ a single general-purpose relation scheme - generally referred to as a *universal triple store* [Sharman and Winterbottom 1988] - of the form:

```
(SUBJECT-ENTITY RELATION OBJECT-ENTITY)
```

where each tuple (or *triple*) of such a relation scheme represents a directed relation between two entities. Any 3-ary or relationship of greater arity can be fragmented into 2-ary relationships that
are storable as triples in such a relation scheme; for example, a tuple in the 3-ary SALE relation scheme:

\[
\text{SALE (BUYER SELLER ITEM-SOLD)}
\]

such as:

\[
(\text{John Smith, Jane Jones, Ming Vase})
\]

- where Jane Jones sells a Ming Vase to John Smith - can be considered as this set of triples:

\[
\begin{align*}
(\text{Sale#1. Buyer. John Smith}) \\
(\text{Sale#1. Seller. Jane Jones}) \\
(\text{Sale#1. Item-Sold. Ming Vase}) \\
(\text{Sale#1. Is-Element. Sale})
\end{align*}
\]

in a universal triple store. Note that, in order to model this non-binary relationship:

- the sale has had to become an explicit entity.
- the sale has had to acquire an additional surrogate (as well as a new triple carrying typing information) in order to distinguish this instance of a sale from other instances.
- the domain roles of the SALE scheme have become relations within the universal triple store.

In the 3-ary approach, the sale surrogate is simply the implicit row ident, and each instance of SALE is a tuple of that relation scheme.

3.2.3.1 Advantages of Binary Relational Approach

The storage strategy of a binary relational system is obviously much simpler than that of an N-ary relational system, and this simplicity of approach has a number of advantages - the most obvious of these is that, because all data can be stored in a single universal triple store, file
design is effectively redundant - and the user needs very little knowledge about the storage implementation because the implementation is extremely simple. In addition, if a binary relational approach is used for both analysis and specification (as would be the case with analysis methods such as entity-relationship modelling), a consistent conceptual framework can be applied throughout - in direct contrast to the more traditional view - and so systems designed using binary relational methods can also be implemented using binary relational storage.

Because the universal triple store outlined above is completely general-purpose, changes in the structure of the application domain do not change its underlying structure - and so application programs require little or no modification to deal with scheme evolution, as new relations are merely different sorts of triples in the universal triple store.

Finally, retrieval from a single relation scheme of constant arity is also easier than retrieval from multiple relation schemes of variable arity. Binary relational retrieval is performed by using one of a small set of simple associative forms, depending on whether a particular position in the triple is known. These associative forms range from FINDTRIPLE (x, ?, ?) to FINDTRIPLE (x, y, z) - from where only the first entity is known, to where all information is known.

3.2.3.2 Disadvantages of Binary Relational Approach

The binary relational approach does also have some drawbacks. Although retrieval of binary relations such as:

(John. Employed-By. ?)

is simple and can be highly efficient, retrieval of more complex, graph-like structures with multiple relations tends to be inefficient. For example, consider the set of triples:


Each triple requires the use of a retrieve operation in order to discover its various instantiations; these retrievals must then be followed by a merging of all the resultant sets. This is generally
conceded to be less efficient than retrieving a tuple or set of tuples from a normalised set of relation schemes using traditional methods.

3.2.4 TRISTARP and Related Advances

Recently, systems such as TRISTARP [Peltu 1994] have begun to appear. They utilise a variety of new indexing techniques to offer improved retrieval speeds and are, in effect, more like content-addressable memories than databases in the usual sense of the word. Universal triple stores have also appeared as general-purpose data stores in several systems - such as the object-oriented Ogetto [Mariani 1992], which uses a universal triple store to record both data and structural metadata (such as hierarchical and class structural information). The ease with which new attributes can be added to general-purpose stores makes them attractive in this context.

3.2.5 Comparison of N-ary and Binary Relational Approaches

Which of these two competing approaches is preferable in the light of the system’s requirements? The binary relational approach seems to bear many resemblances to simple conceptual graphs. It can also store metadata within its general-purpose storage structure, and so is capable of storing the extrarelational information that conceptual graphs can model - in short, its binary relational structure allows storage of simple conceptual graphs. Conversely, the N-ary relational approach seems to be completely limited by the relational model and has no place for storage of this extra information. The binary relational approach also uses a very simple file structure, and its definition and manipulation systems are also very simple; implementation of an application domain is also very simple. Conversely, the more traditional N-ary approach requires normalisation of file structures and detailed knowledge of how they are organised in order to manipulate them.

3.2.6 Early Use and Rejection of Binary Relational Approach

On these grounds, an approach based on binary relational ideas seemed quite promising.
[VEHICLE] - 
(ATTR)→[COLOUR] 
(LOC)→[PLACE] - 
(PLACENAME)→[NAME: “Edinburgh”], 
(POSS)→[TALL-PERSON: #333].

(a) Graph: ‘A Vehicle from Edinburgh Owned By Tall Person #333’

FINDTRIPLE (?1. Is-Element. Land-Vehicle) FINDTRIPLE (?1. Loc. ?2)
... FINDTRIPLE (?1. Poss. ?5)
FINDTRIPLE (?1. Is-Element. Submarine)
FINDTRIPLE (?2. Is-Element. Place)
FINDTRIPLE (?3. Is-Element. Colour)
FINDTRIPLE (?4. Is-Element. Placename)
FINDTRIPLE (?5. Is-Element. Tall-Person)

(b) Triple Retrievals
(c) Triple Merges

Figure 3.2 A Simple Query and Its Binary Relational Retrieval

Early versions of C-GRASS were based upon a universal triple store which could, like the Ogetto system mentioned earlier, store both data and metadata identically (see [Bowen and Kocura 1993] - reprinted in Appendix D - for a description of part of this research). However, two factors conspired against the practical use of such a system:
although data - in the form of conceptual graphs - could be stored in a universal triple store, the binary relational approach has certain organisational conventions which led to problems.

the requirement to make C-GRASS usable with existing relational technology effectively ruled out an approach like that of TRISTARP - this reliance on existing database systems exacerbated the organisational problem.

3.2.6.1 Hierarchical Problems with Binary Relational Strategy

These two factors manifested themselves most spectacularly in the treatment of hierarchical information. In binary relational storage, hierarchical information can be naively stored 'as is', as a series of metadata triples in the universal triple store - so storage of hierarchical information presents no real problems. But whilst both a binary relational store and conceptual graphs can handle hierarchical information, even a trivial hierarchy can cause major problems in retrieval from a universal triple store. To illustrate this problem, consider a simple query such as that found in Figure 3.2, and the fragment of a type hierarchy in Figure 3.3. This is a very simple hierarchy, and yet the query of Figure 3.2 causes major retrieval problems for a universal triple machine in conjunction with that lattice. This is because the type VEHICLE has quite a few subtypes\(^1\) and, in order to generate a complete answer to the query, it is first necessary to find all typings involved, which requires eighteen retrievals (Figure 3.2(b)). Such retrievals find all instances of the types currently of interest. On top of this, the more usual 'untyped' relational retrievals that link these type sets together has to be performed too (Figure 3.2(c)). Using such a strategy - where conceptual graphs in the system are stored as asserted - this simple graph query therefore requires at least twenty-two retrievals in all - this is not including the retrievals required to retrieve the subtypes of VEHICLE from the metadata triples\(^2\). This example also assumes that most of the entities and all of the relationships are primitive - probably unrealistic; and with a realistic domain, retrieval would be unacceptably complex.

\(^1\) The status of COLOUR, PLACE and NAME is not important here, although their possession of subtypes would certainly exacerbate the problem.

\(^2\) We have excluded this from our example as the methods used to calculate such information are unclear; however, we estimate that it would require an additional fifteen retrievals to find all subtypes of VEHICLE from this lattice - first, to perform FINDTRIPLE (Vehicle. Subtype. ?) and then one retrieval for each subtype of Vehicle, and one for each subtype of those subtypes, and so on - fifteen retrievals in this lattice.
3.2.6.2 Encapsulation Problems with Binary Relational Strategy

Hierarchy also introduces a retrieval problem based on encapsulation of data. For instance, consider the graphs of Figure 3.4.

Figure 3.3 Fragment of a Vehicle Hierarchy
(a) 'A Tall and Wealthy Man #1 is the Sibling of a Woman #2'

[TALL-WEALTHY-MAN: #1]→(SIBLING)→[WOMAN: #2].

(b) 'A Man is Wealthy'

[M MAN: #1] -  
  (ATTR)→[TALLNESS]  
  (ATTR)→[WEALTHINESS]  
  (SIBLING)→[WOMAN: #2].

(c) Expanded Form of Figure 3.4(a)

Figure 3.4 A Problem of Encapsulation

Although the individual #1 of Figure 3.4(a) is a tall and wealthy man, the fact that the TALL-WEALTHY-MAN definition contains this information about WEALTHINESS may not be known to the retrieval operation operating over Figure 3.5(b), and so it would fail to retrieve it. If Figure 3.4(a) were expanded however, the retrieval would see the graph of Figure 3.4(c), which would be retrieved.

As well as giving a solution to this encapsulation problem, the operation of expansion also looked as though it might useful in tackling other hierarchically-based problems. This observation led to the development of an alternative scenario in which asserted graphs are stored in their most primitive form by the application of maximal expansion. For example, the graph of Figure 3.2(a) would maximally expand to Figure 3.5.
This means that only the most primitive triples need to be retrieved when executing FINDTRIPLES, but the use of expansion greatly increases the size of the query graph to be processed - and so the number of triples that need to be retrieved and merged is also increased.

FINDTRIPLE (?1. Is-Element. Vehicle)
FINDTRIPLE (?2. Is-Element. Place)
FINDTRIPLE (?3. Is-Element. Colour)
FINDTRIPLE (?4. Is-Element. Placename)
FINDTRIPLE (?5. Is-Element. Person)
FINDTRIPLE (?6. Is-Element. Tallness)
FINDTRIPLE (?1. Attr. ?3)
FINDTRIPLE (?1. Loc. ?2)
FINDTRIPLE (?2. Placename. ?4)
FINDTRIPLE (?1. Poss. ?5)
FINDTRIPLE (?5. Attr. ?6)
If the maximally expanded query given above is used, only eleven retrievals (see Figure 3.6) are required - in comparison with at least twenty required in the non-expansive strategy.

3.2.6.3 Maximal Expansion in Binary Relational Strategy

A maximally expansive strategy is preferable to a non-expansive strategy because:

- it solves the encapsulation problem easily.
- the size and complexity of an application domain’s lattices have very little effect on the number of relationships and types found in storage - only primitive relationships generate relation schemes, and their number tends to grow slowly.
- working out the subtype set of a type is unnecessary because all types in a query graph are primitive.
- it only needs to perform one membership retrieval for each concept in the query.
- because only primitive retrievals are undertaken, there is no requirement to merge sets of triples that are hierarchically related. For example, the merging of sets like:

```plaintext
FINDTRIPLE (?1. Is-Element. Vehicle)
FINDTRIPLE (?1. Is-Element. Land-Vehicle)
FINDTRIPLE (?1. Is-Element. Water-Vehicle)
FINDTRIPLE (?1. Is-Element. Wheeled-Land-Vehicle)
...
```

would be unnecessary because such hierarchical sets do not occur if an expansive strategy is being used.

3.2.6.4 Disadvantages of Maximal Expansion

However, a maximally expansive strategy does have a number of disadvantages as well:
- expansion is an overhead in both storage of a graph and querying - it is a step that needs to be done in order to get graphs into a usable form.
- expansion may lead to more concept nodes and thus to more membership retrievals - this tends to offset some of those retrievals saved above.
- expansion may lead to more dyads, which means that there will be more sets of triples to be merged together - although such set merges are primitive. For instance, the graph of Figure 3.2(a) maximally expands to the graph of Figure 3.5, and so has one more dyad to merge than the original.

3.2.7 Adoption of N-ary Relational Approach

The introduction of hierarchical information into a domain causes an explosion in the number of relational operations required by any binary relational approach, and this factor is the primary reason for the abandonment of the approach shown in [Bowen and Kocura 1993] (see Appendix D).

However, is an N-ary relational strategy any better? It can be argued that, since a binary relational strategy uses general-purpose storage structures, it can store any data - including hierarchical information - whereas an N-ary relational strategy is truly flat and unable to model any hierarchical information at all³. Given this fact, how can N-ary relational storage handle hierarchical information?

3.2.7.1 Hierarchical Problems with Naive N-ary Relational Strategy

The simple answer to this question is that a flat structure cannot easily model hierarchical relationships without additional structures to help it - in C-GRASS, it is assumed that additional lattice structures exist for this purpose.

The hierarchical problems of N-ary relational storage are not really the same as those faced by binary relational storage - in the latter, the problem is more operational; in the former, more

³ Although various ingenious systems such as [Jagadish 1989] and [Kung 1990] offer some form of approach to this problem.
structural. The difference between the two is that whereas the latter approach relies on a single storage structure and query complexity is related to lattice size and complexity, an N-ary relational approach also has storage problems - as the number of storage structures required by the former is also directly related to the size and complexity of an application domain’s type and relation lattices.

Figure 3.7 *Some Vehicles and Their Locations*

(a)

(b)

(c)
If the fragment of hierarchy shown in Figure 3.3 is again used, consider how the graphs of Figure 3.7 would be stored in an N-ary relational system. Figure 3.7(a) is composed of primitive types and relations, and would require the following normalised schemes to store individual graphs of that particular shape:

S1   (VEHICLE COLOUR PLACE)  
S2   (PLACE NAME)

However, a tank (Figure 3.7(c)) and a submarine (Figure 3.7(b)) can’t be put in these relation schemes without losing information - specifically, the properties that make individual #5 a submarine and individual #12 a tank. In order to preserve this information, some more relation schemes are needed:

S3   (SUBMARINE COLOUR PLACE)  
S4   (TANK COLOUR PLACE)

If a graph dealing with a hovercraft now happen to asserted, where does it get stored? It should by now be obvious that each subtype requires its own relation scheme, so a graph like the others containing a hovercraft would need:

S5   (HOVERCRAFT COLOUR PLACE)

For the simple graph above therefore, many relation schemes are needed - the number of which is dependent on the complexity of the hierarchy.

3.2.7.2 N-ary Relational Strategy and Some Hierarchical Assumptions

The development of the schemes shown here has relied on four assumptions:

- that VEHICLE is the only concept label with any subtypes - almost certainly something that cannot be relied upon.
- that the relation hierarchy is primitive - again, almost certainly an unreliable assumption.
that, for instance, a LAND-VEHICLE will always be a LAND-VEHICLE and not become something else - such as a TANK - if new information arises.

- that all asserted graphs are the same shape - for instance, all graphs ever asserted will be of the form seen in Figure 3.8.

Each of these assumptions can be dealt with in turn.

First, it is quite possible for a label like PLACE to have a subtype such as PLACE-BUILT. In such a scenario, each existing relation scheme containing PLACE as a domain would have to spawn a scheme featuring PLACE-BUILT instead:

S1A  (VEHICLE COLOUR PLACE-BUILT)
S2A  (PLACE-BUILT PLACENAME)
S3A  (SUBMARINE COLOUR PLACE-BUILT)
S4A  (TANK COLOUR PLACE-BUILT)
...

If COLOUR and PLACE have multiple subtype, the number of relation schemes required to store all graphs without loss of information is highly explosive.
[TANK: #12] -

(ATTR)→[GREEN-COLOUR]

(HOME-LOC)→[PLACE: #23] -

(PLACENAME)→[NAME: "Fulda"]..

Figure 3.9 'A Green Tank From Fulda'

Second, it is entirely possible for conceptual relationships to be nonprimitive - (FATHER-OF) and (MOTHER-OF) as subrelations of (PARENT-OF), for instance. If (LOC) possessed the subrelation (HOME-LOC) in the above example, how would the graph of Figure 3.9 be stored? This graph can't be stored in any of the schemes that have gone before, as information about the relationship between the TANK and the PLACE would be lost. Once again, the only answer is to generate yet another relation scheme:

\[
S4B \quad (\text{TANK COLOUR PLACE})
\]

that happens to be identical to S4, but maps to a different conceptual graph in main memory.

Third, what happens if information is asserted that allows the instance #8 (a VEHICLE) to be reclassified as a TANK? Information about instance #8 is currently resident in those relation schemes wherein other VEHICLE instances are stored, and so tuples featuring instance #8 would have to be moved from those schemes into their equivalent TANK schemes; this is quite a complicated little overhead that arises purely from the hierarchical partitioning of relation schemes.

Finally, it has been assumed that all assertions will look the same; the possibility of different graphs such as that of Figure 3.10 has not been admitted. It might seem that this can be solved by just mapping the new area of graph to the relation scheme:

\[
S6 (\text{PLACE NAME})
\]
However, N-ary relational storage relies on normalisation to eliminate data redundancy, and it may well be that the new information is mandatory for each vehicle (and its subtypes). This scenario would necessitate existing scheme restructuring in order to accommodate new information:

\[
\begin{align*}
S1 & \quad (\text{VEHICLE COLOUR PLACE-1 PLACE-2}) \\
S2 & \quad (\text{PLACE-1 NAME}) \\
S3 & \quad (\text{SUBMARINE COLOUR PLACE-1 PLACE-2}) \\
\ldots & \\
S7 & \quad (\text{PLACE-2 NAME})
\end{align*}
\]

### 3.2.7.3 Hierarchy and Retrieval in an N-ary Relational Strategy

As if these massive storage and maintenance problems aren't enough, retrieval in such a system may not be simple either: this non-expansive strategy is as prone to the encapsulation problem as its binary relational counterpart, and simple querying is badly affected by the hierarchical partitioning of schemes. For example, consider the graph of Figure 3.2(a) again. For ease of description, this graph will be broken into subgraphs relating to relation schemes (as described in the earlier chapter on mapping conceptual graphs to the relational model) - these subgraphs are shown in Figure 3.11.
Each subgraph requires the union of the scheme to which it maps with the various hierarchically-partitioned subschemes, before being joined together. In SQL, something like the following set of commands would produce an answer to the query shown in Figure 3.12.

Figure 3.11(a) is primitive and is a simple select; Figure 3.11(b) is hierarchically partitioned into fourteen schemes so requires fourteen selects and thirteen unions; Figure 3.11(c) is similarly partitioned into fourteen schemes and so needs fourteen selects and thirteen unions. On top of this, two equijoins are required to link the various views together properly. Frankly, this is not efficient at all, and the use of the relational union operator also gives rise to another problem - ensuring that the schemes being unioned together are of equal arity; guaranteeing this can impose a number of additional problems.⁴

⁴ See [Bowen and Kocura 1994] (which appears in Appendix E) for a more in-depth treatment of this issue.
CREATE VIEW SG2 (PLACE, NAME) AS
SELECT PLACE, NAME FROM S2 WHERE NAME = "Edinburgh";

CREATE VIEW SG1 (VEHICLE, COLOUR, PLACE) AS
SELECT VEHICLE, COLOUR, PLACE FROM SCHEMEW UNION ...
SELECT SUBMARINE, COLOUR, PLACE FROM SCHEMEX

CREATE VIEW SG1 (VEHICLE, TALL_PERSON) AS
SELECT VEHICLE, TALL_PERSON FROM SCHEMEY
WHERE TALL_PERSON = "#333" UNION ...
SELECT SUBMARINE, TALL_PERSON FROM SCHEMEZ
WHERE TALL_PERSON = "#333";

SELECT A.VEHICLE, A.COLOUR, A.PLACE, B.NAME, C.TALL_PERSON
FROM SG1 A, SG2 A, SG3 C
WHERE A.PLACE = B.PLACE AND A.VEHICLE = C.VEHICLE;

Figure 3.12 Retrieval of Figure 3.2 in SQL

3.7.2.4 Impracticality of Non-Expansive N-ary Relational Strategy

A non-expansive implementation of hierarchy in N-ary storage can be considered as worse than its corresponding non-expansive implementation in binary relational storage, because it does not make the most of the major advantage of the N-ary relational approach - superiority in retrieval efficiency. Because relation schemes are hierarchically partitioned to avoid information loss, a large number of small but specialised schemes are generated; indexing of such schemes is of small benefit, and producing a comprehensive answer to a query may take an awful lot of scheme merging/unioning.
However, use of an expansive strategy in N-ary relation storage is very different and is certainly superior. It is based on creating relation schemes based on only the most primitive types and relations\textsuperscript{5}, and using maximal expansion during retrieval. Hierarchical problems are therefore eliminated by flattening the schemes and pushing any hierarchical processing back to the graph level. The removal of explicit hierarchical information from the storage level tends to lead to a much smaller number of larger relation schemes (in terms of the number of tuples stored in each, that is) than is possible in a strategy where hierarchical scheme partitioning is used - this is because all graphs are expanded into primitive dyads, so giving rise to large numbers of them.

\textsuperscript{5}Primitive types do not include the universal type and relation (which gives us virtually no information); maximal expansion therefore refers to expansion into the immediate subtypes and relations of the most general lattice nodes.
As an example, consider the graph of Figure 3.13(a), which can be stored in the normalised schemes:

\[
\begin{align*}
S1 \text{ (VEHICLE COLOUR PLACE)} \\
S2 \text{ (PLACE NAME)} \\
S3 \text{ (VEHICLE PERSON TALLNESS)}
\end{align*}
\]

whilst the graph of Figure 3.13(b) - which would require generation of the scheme:

\[
S4 \text{ (VEHICLE TALL-PERSON TALLNESS)}
\]

using a non-expansive strategy - can be stored in exactly the same schemes as it maximally expands into a graph with identical primitive types and connections as the previous graph - a new relation scheme is not needed to store it, as the hierarchy is no longer explicitly partitioning relation schemes. Therefore, the generation of schemes is no longer potentially explosive. This removal of partitioning also means that no scheme unions are required, although expansion gives rise to more primitive dyads and so more joins may be required;

Because only small sets of large cardinality schemes are used, efficient indexing can be exploited in retrieval, and so make use of the major advantage of N-ary relational storage.

3.2.8 Summary of Storage Issues

Any approach using a non-expansive strategy tends to encounter large overheads when retrieving information (and in the case of the N-ary relational approach, explosively generating inefficient relation schemes). These two approaches can be ruled out at this early stage.

3.2.8.1 Suitability of N-ary Relational Approach for Pure Data Storage

As regards the remaining expansive strategies, both have advantages and disadvantages - whilst binary relational storage makes for easier storage, its retrieval performance is less good. Conversely, N-ary relational storage has better retrieval performance but is harder to store
information in. If speed of retrieval is more important than speed of assertion, then the latter approach is preferred, and this is the case in C-GRASS.

3.2.8.2 'Recasting' of Primitive Conceptual Relations

There is perhaps only one serious implementational (as opposed to theoretical) drawback with the primitives-only hierarchical approach, and that occurs upon what might be termed the recasting of a primitive relation or type. Consider the following scenario: the user has asserted the relation (GRANDFATHER) as a primitive relation, and so storage can be based upon it. What now happens if the user decides that (GRANDFATHER) is not a primitive relation after all - it might be that:

(FATHER) > (GRANDFATHER)

The definition for (FATHER) is:

$$FATHER = \lambda x, y \text{ [PERSON: } ^*x{\rightarrow} \text{ [CHILD]} \rightarrow \text{ [MAN : } ^*y \].$$

The nub of the problem here is that (GRANDFATHER) already has storage structures based on its primitive status; these structures may even contain data. The trouble is that:

- the tuples of (GRANDFATHER) no longer conform to a primitive treatment of (GRANDFATHER).
- (GRANDFATHER) is now non-primitive and has no definition.

Therefore, C-GRASS must:

- recompute optimal storage to take account of both the erasure of (GRANDFATHER) and the addition of (FATHER).
- create one or more relation schemes (FATHER), and adapt the mappings accordingly.
- get a definition for (GRANDFATHER) from the user as quickly as possible.
• expand the definition to its maximal form and pad out any non-instantiated tuple attributes.
• store the expanded data in the new schemes (possibly including the new (FATHER) locations).

Although this situation is unlikely to occur very frequently, C-GRASS takes account of the possibility.

3.2.8.3 Binary Relational Approach as a 'Repository'

Although N-ary relational storage is used for pure data storage, the binary relational approach has not been completely abandoned. Although it isn't really suitable for storage of large data sets, it is eminently suitable for storing smaller sets of data that carry more semantics, and it fulfils this role in C-GRASS.

When a C-GRASS session is terminated, the user presumably does not wish to lose the information that has been asserted in that session. Although the data has been safely stored away, the meaning of that data - the conceptual graphs and the mappings between them and the relational structures - will be lost; in order to avoid this, C-GRASS has two repository schemes - one for the storage of contextual information, and the other for the storage of the simple graphs within those contexts. As the data-free conceptual graphs of the application domain are dyadic and are not very large, this second repository scheme uses an approach based on binary relational storage to store the basic dyads and their associated mappings. The binary relational approach is therefore useful to C-GRASS as a metastore, rather than a data store - it stores semantics rather than data.

3.3 Database Issues II: Adaptive Storage

Having discussed the problems related to fitting heterogeneous conceptual graphs into homogeneous structures - and showing that information that is traditionally regarded as outside
the realm of the relational model can be dealt with - the next important issue is that of efficient storage and retrieval of conceptual graph structures in a relational environment.

Bearing in mind that efficient retrieval is based on producing connected hypergraphs in which the hyperedges are of maximal size (the larger the hyperedges, the fewer the number of joins required to form a hypergraph corresponding to a conceptual graph in an N-ary relational storage strategy), the task of maximising the size of hyperedges is a subproblem of normalisation, and so some form of database design is necessary if efficient storage and retrieval is to be achieved.

It should also be borne in mind that a research aim is to relieve the end-user of the need to possess implementation-specific knowledge - that is, knowledge of how a particular application domain is stored in database terms and how it may be manipulated. Whilst this aim is geared primarily towards removing this requirement in the area of querying, it applies equally well in the area of file design and maintenance. If the aim of shielding the user from the need to possess implementation-specific knowledge is to be satisfied, file design and evolution obviously cannot be performed by the user; this means that the responsibility for file design and evolution must rest with the system. This section deals with the performance of automated normalisation in C-GRASS.

3.3.1 Cardinality and Membership in Adaptive Storage

Earlier, it was mentioned that cardinality and membership information could be extracted from an application domain and used to maintain the relation schemes used in a given application domain at some level of normalisation. C-GRASS aims to keep its relation schemes in at least the third normal form, and there are a number of steps that need to be performed in order to achieve this.

3.3.1.1 Storage for Primitive Relations

The first step is concerned with the production of basic default storage. As C-GRASS uses a maximally expanded storage strategy, the only relationships that require storage and mappings
are the most primitive relationships. C-GRASS enforces a condition that any assertion must be canonical in order to be stored, which is in line with regular conceptual graph theory. However, a subcondition is also enforced - that all primitive relationships must have an entry in the canonical model; because any assertion will be maximally expanded into these primitives, this subcondition ensures that any assertion will be completely storable. Although only the most primitive relationships are extracted, there may well be more than one instance of each extracted. For example, the dyads:

\[\text{[PERSON]} \rightarrow \text{(GENDER)} \rightarrow \text{[MALENESS]}\].

and:

\[\text{[PERSON]} \rightarrow \text{(GENDER)} \rightarrow \text{[FEMALENESS]}\].

are both primitive in both relation and conceptual attachments, yet both are different and would require different relation schemes in order to represent both of them. Any primitive relationships that are extracted from the canonical model are deemed to have a default N-M cardinality with optional membership on both arcs.

3.3.1.2 Specialisation of Default Cardinality and Membership

The next step is to try to specialise these rather crude defaults into something that is of more interest in schema design, and so the graphs of the application domain are checked for constructions that indicate a specialisation of cardinality - the theoretical basis for this extraction has been previously given. After this, an attempt to restrict the optional memberships of the primitive dyads extracted from the canonical model is made; this is done by searching for Skolem-like constructions in the graphs of the application domain.

When looking for such constructions, it is important to note that the antecedent of the graph must not be more specialised than the target concept - for example, the graph of Figure 3.14(a).
( [CAR: *x]→(COLR)→[GREEN] ( [CAR: *x]→(OWNER)→[PERSON] ) )

(a)

( [CAR: *x] ( [CAR: *x]→(OWNER)→[PERSON] ) )

(b)

Figure 3.14 Graphs for the Extraction of Memberships

This graph wouldn’t be acceptable for deriving mandatory membership on CAR, because it isn’t on all cars, just those which are green. The consequent can be more specialised, so long as it contains the core linkage mentioned above. If such a construction can be found for a relationship, membership is specialised from optional to mandatory. For example, the graph of Figure 3.14(b), in which the membership of CAR when participating in the (OWNER) relationship - assuming that (OWNER) is primitive, of course - will become mandatory.

3.3.1.3 Comparison of Adaptive Information Against Previous Knowledge

The next step tests the newly-extracted relationship set against a previous set - which resulted from the last domain modification.

If the new set contains any relationships that are foreign to the old set, then such relationships are a result of the latest change in the application domain, and a new relation scheme and appropriate mappings must be created for each.

If any relationships appearing in the old set appear in the new set, yet have different membership or cardinality properties, then existing relation schemes may well change. In order to find out if
any relation schemes need changing, C-GRASS searches the new list for posting events\(^6\) and normalises the primitive schemes wherever possible by posting. This is a classical technique of entity-relationship diagrams and, as such a diagram has basically been distilled from conceptual graphs in the form of a set of cardinalities and memberships, such a technique is applicable here. Posting essentially involves merging the contents of an entity (a concept in this case, and the things dependent on it) or a relation into another entity.

3.3.1.4 Schema Restructuring: Addition, Modification and Deletion

Based on the results of a search for cardinalities, new relation schemes may be created that merge two or more old relation schemes together, or an old relation scheme may be split into two or more new schemes. In addition to the creation of these new schemes, C-GRASS also calculates how the data present in the subsumed old schemes may be mapped into the new scheme, before moving that data and deleting the now-redundant old schemes. An important point worth noting here is that if a cardinality violation results from transferring the data into the new scheme, the transfer is prevented and the user warned about the violation. For example, say that a relation scheme containing data about spouse relationships contains the information that Billy Bigamist is married to more than one wife; if the spouse relationship was N-M, this wouldn’t matter much. If changes to the application domain force the spouse relationship into a 1-1 cardinality, the change in the application would be prevented as Billy’s two spouses would violate that cardinality if allowed to exist.

If a relationship existing in the old set doesn’t exist in the new set as well, then that relationship has presumably disappeared from the application, and so the relation schemes that map to it may also be removed - this may only occur if those schemes are empty.

\(^6\) Again, see Chapter 2 and Appendix A for details of these situations and the reasoning behind them.
3.3.1.5 Structure of Cardinality

Keeping cardinality and membership information explicitly in the canonical model and application graphs would lead to inefficiency when restructuring of normalised storage is required, and so C-GRASS maintains this information in an accelerating structure\(^7\) that is here referred to as a *cardinality*. At its conceptually simplest, each cardinality is a dyadic graph of form \([f \rightarrow (r) \rightarrow t]\), where \(f\) is the 'From' concept from which \(r\) originates, and \(t\) is the 'To' concept to which \(r\) points. As well as the usual referents, both \(f\) and \(t\) also possess domain mappings and a membership; finally, each cardinality object has an overall cardinality (1-1, etc).

3.3.1.6 Computation of Normalised Storage

Recomputation of normalised storage is mediated by Algorithm 3.1, which is unique to C-GRASS. It takes two sets of cardinalities - an old set (upon which existing storage and mappings are based) and a new set (which exists because of changes to the application domain) - that shall be referred to as \(X_{\text{old}}\) and \(X_{\text{new}}\) from this point onwards.

---

**Algorithm 3.1 Computing Normalised Storage:** Given the cardinality sets \(X_{\text{new}}\) and \(X_{\text{old}}\), the computation of a new set of normalised relation schemes for record storage is as follows.

```
for each \(c \in X_{\text{new}}\) do
    create default binary domain mappings for \(c.f\) and \(c.t\)
enddo

for each \(c \in X_{\text{new}}\) do
    if the cardinality of \(c\) is not N-M then
        for each \(m \in X_{\text{new}}\) do
```

---

\(^7\) Similarly, a lattice is an accelerating second-order structure for information that could be represented as complex conceptual graphs - such structures speed up processing.
if the cardinality of m is not N-M then
  if m ≠ c then
    if c.f = m.f and both have obligatory membership then
      merge mappings of c and m;
    else
      if c.f = m.t and both have obligatory membership then
        merge mappings of c and m;
      endif
    endif
  endif
endif
endif
enddo
for each m ∈ Xnew do
  if the cardinality of m is not N-M then
    if m ≠ c then
      if c.t = m.f and both have obligatory membership then
        merge mappings of c and m;
      else
        if c.t = m.t and both have obligatory membership then
          merge mappings of c and m;
        endif
      endif
    endif
  endif
endif
endif
endif
enddo
endif
enddo

use modified \( \chi_{\text{new}} \) to create actual relation schemes;
transfer tuples from old relations of \( \chi_{\text{old}} \) construction to the new relations just created;
if any cardinality violations detected as a result of transfer then
  terminate scheme restructuring by rollback;
endif

copy \( \chi_{\text{new}} \) to \( \chi_{\text{old}} \).

This algorithm computes the normalised schemes required to store \( \chi_{\text{new}} \), and compares them with the existing schemes derived from \( \chi_{\text{old}} \). It then rebuilds the secondary storage, using the differences between \( \chi_{\text{old}} \) and \( \chi_{\text{new}} \) to transfer tuples from the existing schemes to new ones derived from \( \chi_{\text{new}} \). Tuples from old schemes may be joined together in a new scheme or split over a number of smaller schemes, depending on how \( \chi_{\text{old}} \) and \( \chi_{\text{new}} \) differ; and if any violations in cardinality are noted at this transfer stage (for example, an N-1 cardinality in \( \chi_{\text{old}} \) might have become a 1-1 cardinality in \( \chi_{\text{new}} \), but not all the existing data agrees with this new cardinality), a warning is issued and a rollback initiated to prevent any of the changes just made.

The heart of this algorithm is the merging of mappings, and this requires some explanation. All members of \( \chi_{\text{new}} \) are initially allotted a default scheme name and mappings, and each default is different for each member. Whilst these defaults could be used as a basis for generation of storage, it would not produce a particularly efficient set of relation schemes.

As \( \chi_{\text{new}} \) is a series of cardinalities and memberships, it is actually analogous to an entity-relationship diagram, and the technique that is being used here to merge mappings is based upon classical entity-relationship posting - this essentially involves merging the contents of an entity
(a concept in this case, and the things dependent on it) or a relation into another entity. The basic strategy is to scan each member of \( \chi_{\text{new}} \), and to test both the \( f \) and \( t \) of any non-many-to-many cardinalities for attachments to any other non-many-to-many cardinalities.

Referring back to \( c \) and \( m \) of Algorithm 3.1, there are basically four ways in which these structures can link together (Figure 3.15).

i. a join on \( cf \) and \( mf \) (for example, Figure 3.15(a)). If a join gets \( T(\text{true}) \) from a scan of this matrix:

\[
\begin{array}{ccc}
  & 1-1 & 1-N & N-1 \\
1-1 & T & F & T \\
1-N & F & F & F \\
N-1 & T & F & T \\
\end{array}
\]

\[
\begin{array}{c}
  c \quad \text{[BOOK]} \rightarrow (\text{ATTR}) \rightarrow [\text{COPY}] \\
  m \quad [\text{BOOK}] \rightarrow (\text{ATTR}) \rightarrow [\text{ISBN}] \\
  c \quad [\text{PERSON}] \rightarrow (\text{AGE}) \rightarrow [\text{NUMBER}] \\
  m \quad [\text{SHOE}] \rightarrow (\text{SIZE}) \rightarrow [\text{NUMBER}] \\
\end{array}
\]

(a) \hspace{7cm} (b)

\[
\begin{array}{c}
  c \quad [\text{COPY}] \rightarrow (\text{LOC}) \rightarrow [\text{PLACE}] \\
  m \quad [\text{BOOK}] \rightarrow (\text{ATTR}) \rightarrow [\text{COPY}] \\
  c \quad [\text{BOOK}] \rightarrow (\text{ATTR}) \rightarrow [\text{COPY}] \\
  m \quad [\text{COPY}] \rightarrow (\text{LOC}) \rightarrow [\text{PLACE}] \\
\end{array}
\]

(c) \hspace{7cm} (d)

Figure 3.15 Four Ways to Join Cardinality Structures
then a merge of the two scheme names associated with the two cardinalities is possible -
but only if the membership test is also passed.

ii a join on $c.t$ and $m.t$ (for example, Figure 3.15(b)). If a join gets $\text{T}(\text{rue})$ from a scan of
this matrix:

\[
\begin{array}{ccc}
1-1 & 1-N & N-1 \\
1-1 & T & T & F \\
1-N & T & T & F \\
1-N & F & F & F \\
\end{array}
\]

then a merge of the two scheme names associated with the two cardinalities is again
possible - but only if the membership test is also passed.

iii a join on $c.f$ and $m.t$ (for example, Figure 3.15(c)). If a join gets $\text{T}(\text{rue})$ from a scan of
this matrix:

\[
\begin{array}{ccc}
1-1 & 1-N & N-1 \\
1-1 & T & F & T \\
1-N & T & F & T \\
N-1 & F & F & F \\
\end{array}
\]

then a merge of the two scheme names associated with the two cardinalities is again
possible - but only if the membership test is also passed.

iv a join on $c.t$ and $m.f$ (for example, Figure 3.15 (d)). If a join gets $\text{T}(\text{rue})$ from a scan of
this matrix:

\[
\begin{array}{ccc}
1-1 & 1-N & N-1 \\
1-1 & T & F & T \\
1-N & T & F & T \\
N-1 & F & F & F \\
\end{array}
\]
then a merge of the two scheme names associated with the two cardinalities is again possible - but only if the membership test is also passed.

The membership test depends on the things being joined - if a 1-1 cardinality is involved in joining with another 1-1 cardinality, then all the memberships in the join must be mandatory; otherwise, any N-cardinals in the cardinalities being joined must be mandatory in joins featuring N-1 or 1-N cardinalities. It seems to be a general rule that a join between two cardinalities only occurs on a unique key concept - the theory behind this may well be interesting if developed further.

If a join of two cardinalities \( c \) and \( m \) is finally mandated after all of this, then the scheme mapping of \( m \) is recast to agree with that of \( c \), and the attributes to which \( c \) and \( m \) map are also recast so that only the 'join' concepts of \( c \) and \( m \) share the same attribute mapping.

3.3.1.7 **Circumstances Under Which Adaptation Occurs**

The adaptive storage test needs to be performed whenever there has been a change to the application that might result in a change to the cardinality and/or membership of any relation scheme in the underlying storage system - after a change to the canonical model, after a change to the main graphs of the application domain that are separate from the canonical model, or if a primitive relationship is 'recast' to nonprimitive status.

3.4 **Conceptual Graph Issues I: Graph Structures in C-GRASS**

Having discussed the database design issues of C-GRASS, the next important step is towards more explicitly graph-oriented design issues. The first major issue that must be addressed is simple - what sort of structures is C-GRASS able to operate upon?

C-GRASS supports seven sorts of object that can be manipulated by the user: the type and relation hierarchies, type and relational definitions, and two sorts of first-order conceptual graph - the application graph (which is a set of graphs containing all the rules and semantics of
the application domain, plus mappings to areas in secondary storage) and the canonical model (a graph that acts as the upper and lower bounds of what is canonical in the domain). As C-GRASS does not support actors, an alternative form of computational structure - the type-like function - is also supported as a structure.

3.4.1 Type Hierarchy

The type hierarchy in C-GRASS is almost the same as found in [Sowa 1984], but it contains two important modifications - the ability to support non-Aristotelian arcs (which are refer to as logical rather than Aristotelian arcs), and the support of 'metatypes'.

3.4.1.1 Two Types of Hierarchical Arc

The reasoning behind the support of logical arcs is grounded in an attempt to establish a position on the tricky matter of multiple supertypes. For example, consider a scenario (Figure 3.16) where the types WATER-VEHICLE, LAND-VEHICLE and AMPHIBIOUS-VEHICLE are subtypes of VEHICLE. This in itself causes no problem, although it would be sensible to make AMPHIBIOUS-VEHICLE a subtype of both WATER-VEHICLE and LAND-VEHICLE; the problem that arises from this situation is one of priority in definition - which type (LAND-VEHICLE or WATER-VEHICLE) should be used as a genus type for the definition of AMPHIBIOUS-VEHICLE?

The C-GRASS solution (Figure 3.17) is to introduce non-Aristotelian arcs between both LAND-VEHICLE and AMPHIBIOUS-VEHICLE, and between WATER-VEHICLE and AMPHIBIOUS-VEHICLE. Such non-Aristotelian arcs permit inheritance only in the most general sense - inheritance of supertype properties is not permitted, only the fact that a type is a subtype. This allows VEHICLE to provide the genus type (as it is the only type with AMPHIBIOUS-VEHICLE as a direct Aristotelian subtype), whilst still allowing AMPHIBIOUS-VEHICLE to be a subtype of both LAND-VEHICLE and WATER-VEHICLE.
3.4.1.2 Metatyping of Concepts

Although mechanisms already exist in regular conceptual graph theory that allow type and relation *labels* to be checked, there is no mechanism for checking that type *referents* are well-formed. This is of importance if the general classes of referents allowed in processing is to be extended beyond the usual referents currently available.

For example, there is nothing to stop us asserting both:
and:

[PERSON: "Martin Chuzzlewit"].

- even though it is naively known that concepts of type PERSON should always contain individual markers. Similarly, instances of the concept:

[QUANTITY-OF-ITEM].

should always be numeric, but how is such a property to be enforced? An ignorant or careless user might forget to use the @ symbol when forming the referent, for example. In short, the user cannot be relied upon to get it right every time, and they need help if errors are to be avoided.

In pursuit of this aim, C-GRASS introduces the idea of metatyping of concepts. In essence, the metatype of a concept is an additional property of a concept that restricts what may be legally used as a referent for that concept.

In order to preserve the metatype of the referents of a type, a function class is postulated that is similar to the function type:

Definition 3.1 Classes of Types: the function class restricts concepts and their referents by the following rules:

- The types $a$ and $b$ are said to be of the same class if $\text{class}(a) = \text{class}(b)$.
- Any type $t$ may belong to a single class only.
- If the type of concept $c$ is of the individual class, then any non-generic referent of $c$ must be of the form #number.
- If the type of concept $c$ is of the numeric class, then any non-generic referent of $c$ must be of the form number.
• If the type of concept $c$ is of the *string* class, then any non-generic referent of $c$ must be of the form ‘*string*’.

• the class property is inheritable through subtype links.

For example, PERSON-NAME and its subtypes would belong to the string metatype, and constructions like:

```plaintext
[PERSON-NAME: “Mrs Malaprop”].
```

would be legal, whilst constructions like:

```plaintext
[PERSON-NAME: #32556].
```

would not. Metatyping is additionally, therefore, an enforcement mechanism for the difference between lexical and nonlexical objects.

### 3.4.1.3 Deliberate Limitations of Metatyping

It is immediately obvious that the number of metatypes has been restricted to just three, and that only three sorts of non-generic referent are legal under this assumption. There are two good reasons for this limitation. The first reason is related to the issue of database storage. When creating storage structures in the relational model, it is necessary to classify each attribute in the structure by the sort of information that is to be stored in it. Generally, two sorts of information can be stored - either a numeric value of some sort (integer, float, etc) or a string value of some specified length. In order to construct these storage structures without the user’s intervention, the above class information is needed; without it, operations such as range comparison (which operate between numeric attributes) will not work properly. The second reason is concerned with the confusion existing over the rather unfortunate and regrettable trend towards increasing the semantic content of the referent field ([Tjan et al 1989] is a good example of this trend). As in the database view, referents should really be nothing more than *surrogates* - a way of representing an existing entity inside the computer; however, what is generally acceptable as a referent is currently rather confusing.
For instance, [Sowa 1993] lists the constructions of Figure 3.18 as legal. Leaving aside the logical sensibilities of such constructions, this proliferation of symbols in the referent field makes concepts using them difficult to operate upon - for example, how does one join:

\[
\text{[CAT: @} @ 1 \text{]}.
\]

to:

\[
\text{[CAT: (}] @ 2\)\text{]}.
\]

algorithmically - not by hand. Because there is so much scope for confusion, a curb on most of the above notation is needed if an implementation is to succeed, and a restriction to three simple referent types is guided by the wish for such a curb.

In summary then, metatypes allow the extension of individual markers to account for numbers and strings (dates were also considered, as well as floats - they may be introduced at a later date). As relations have no explicit markers, they do not require meta-information, and so (UNIVERSAL) is the top 'effective' type in the relational lattice - as opposed to the metatypes in the type lattice.

\footnote{[Kocura 1990] has been foremost in repudiating the move towards increased semantics in the referent field.}
3.4.2 Relation Hierarchy

Although not explicitly referred to in [Sowa 1984], the need for such a structure is widely acknowledged. The structure supported by C-GRASS is the usual relation hierarchy of conceptual graph theory; because there is no formally established principle of inheritance in a hierarchy of relations - unlike the Aristotelian inheritance in a hierarchy of types - all arcs in the relation hierarchy are what has previously been referred to as logical arcs in the discussion of the type hierarchy.

3.4.3 Type and Relation Definitions

In C-GRASS, the mechanisms for type and relation definition is much the same as in [Sowa 1984] - except that the body of each definition is limited to a single simple graph.

3.4.4 Canonical Model

In the user manual for C-GRASS (Appendix B), the canonical model is often referred to as the second-order graph, whilst the application graph is often referred to as the first-order graph. These terms are something of a misnomer, as both are first-order graphs - the term arises from the fact that the canonical model is the repository of information that is generally second-order in nature.

The canonical model is the principle repository of selectional constraint information in C-GRASS, and an entry into the application graph will not be permitted if it doesn’t conform to the canonical model. Any application graph that can be formed from application of the canonical formation rules over the canonical model is a canonical graph.

3.4.5 Application Graphs

These graphs contain all the information that the user has asserted as application-specific information, as opposed to selectional constraints like those held in the canonical model. It is composed of two major parts:
- the true set, which consists of all new simple graphs joined together as much as possible on identical concepts.
- the false set, which consists of false graphs - both graphs about simple falsity (for example, 'P is FALSE') and the complex graphs that represent the laws of the application domain (for example, 'if P then Q'). These graphs are true statements about falsity in the domain, so it is more accurate to say that 'it is TRUE that P is FALSE', rather than just 'P is FALSE'.

As C-GRASS is built upon relational storage, it therefore makes sense to strip out data from similar graphs and to store it in relational storage structures (as outlined in the section on the separation of data and semantics in the Introduction). Therefore, the application graphs are simply skeleton graphs which contains mappings between itself and the storage locations where the instantiations of the graphs are kept. This does not affect assertion or inquiry in any way, although display of the application graphs will only show this skeleton.

All graphs in both sets are also maximally expanded - this is a consequence of the storage strategy adopted by C-GRASS in order to ease the design and maintenance of relational storage schemes in a system that deals with hierarchical relationships. As with skeletonisation, the only time the user is aware of this strategy is when the first-order graph is displayed. The reasoning behind such a maximally expanded strategy is explored below.

3.4.6 Function Definitions

Functions are the mechanism by which computation is performed in C-GRASS, and are used to replace the computational actors found in [Sowa 1984] and elsewhere. Functions are special-purpose types, although they have no hierarchical structure or Aristotelian system of inheritance. Although each function has a single concept acting as genus, so such a structure may be theoretically possible - although what the use of such a structure would be is debatable.

The computational ramifications of actors and functions are discussed in a later section.
3.5 Conceptual Graph Issues II: Graph Operations in C-GRASS

Having described the graph structures that C-GRASS uses, how can those structures be manipulated? The operation set of C-GRASS is different from the operation set of standard conceptual graph theory, which includes the canonical formation rules and derived operations such as projection and maximal join, and the encapsulation operations of expansion and contraction. C-GRASS possesses most of these operations, but the user has no direct access to them; instead, a series of macro operations have been designed that manipulate the standard operation set automatically, to certain prespecified ends.

An example of such a macro operation is inquiry, which encapsulates operations such as canonical testing, projection, and display. Such operations simplify user access to conceptual graphs, whilst also restricting the ways in which the domain can be accessed; this leads to domain manipulation that is more secure and robust.

In this section, these macro operations and their algorithms are described, as well as the basic conceptual graph operations upon which they are based.

3.5.1 Basic Conceptual Graph Operations in C-GRASS

Conceptual graph theory has a small but powerful set of basic rules and operations for the manipulation of graphs and related structures. The most important of these operations are:

- the canonical formation rules (copy, restrict, join and simplify).
- the derived formation rules (maximal join and projection).
- the encapsulation operations (contraction and expansion).
- the $\phi$ operator.

Given that the overall macro operations (section 3.5.3) of C-GRASS have use of most of these operations, their algorithmic design and ultimate implementation is important if C-GRASS is to function efficiently. Only the algorithms are examined here, deferring efficiency considerations.
3.5.1.1 **Canonical Formation Rules in C-GRASS**

Of the canonical formation rules, only the Copy rule exists in any explicit form. Apart from this, the Join, Simplify and Restrict rules are bound up in the Maximal Join operation (section 3.5.1.2). The Copy rule in C-GRASS has extensions to copy contexts and lines of identity too.

---

**Algorithm 3.2 Copying a Conceptual Graph:** Copy takes a conceptual graph \( u \) as its argument, and returns a conceptual graph \( v \) that is an exact copy of \( u \):

```plaintext
for each context \( \chi \) in \( u \) do
    make a copy \( X \) of \( \chi \) and add it to \( v \);
enddo
for each \( \chi \) in \( u \) do
    copy the contextual dominations of \( \chi \) to its equivalent \( X \);
enddo
for each \( \chi \) in \( u \) do
    for each graph \( \gamma \) in \( \chi \) do
        for each concept \( c \) of \( \gamma \) do
            make a copy \( k \) of \( c \) and attach it to \( X \);
        enddo
        for each concept \( c \) of \( \gamma \) do
            copy the relationships of \( c \) to \( k \);
        enddo
    enddo
enddo
for each \( \chi \) in \( u \) do
    for each \( \gamma \) in \( \chi \) do
        for each concept \( c \) of \( \gamma \) do
            copy any lines of identity \( L \) possessed by \( c \), to \( k \);
        enddo
    enddo
enddo
```
3.5.1.2 Join

The maximal join operation of C-GRASS encompasses the canonical formation rule of join, along with the rule of simplification (which is used to tidy up the graph afterwards) and restriction (which modifies the labels of concepts when they are being joined). In C-GRASS, maximal join is only permitted between conceptual graphs that both possess graphs at their zero contextual depths, and any joining that occurs will not cross contextual boundaries.

Algorithm 3.3 Maximal Join of Two Conceptual Graphs: Maximal Join takes two conceptual graphs \( u \) and \( v \) as arguments, and returns the maximal join of \( u \) and \( v \), \( w \):

for each concept \( c \) in \( u \) do
  for each concept \( k \) in \( v \) do
    if \( c \) subsumes \( k \) then
      \( c \) and \( k \) are a concept match - try to maximally extend the local join site around it;
      note the size of any local join \( j \) around \( c \) and \( k \);
    endif
  enddo
enddo

for each concept \( c \) in \( u \) do
  while a maximal local join \( j \) for \( c \) has not been found do
    find the largest join from \( \{j\} \) in which \( c \) participates;
  endwhile
enddo
if such a join $j$ exists then
    for each concept $\kappa$ in $j$ do
        for each join $J$ do
            if $\kappa$ appears in $J$ and $|J| < |j|$ then
                $j$ is discarded;
            endif
        enddo
    enddo
endif
endif
endif
if $j$ has not been discarded then
    discard other joins in which $c$ participates, as $c$ has
    the maximal local join $j$ in which it appears;
endif
enddo
enddo
join each concept match between $u$ and $v$ indicated;
simplify $u$ to produce $w$.

3.5.1.3 Projection

Algorithm 3.4 below is the basic projection algorithm for simple conceptual graphs. C-GRASS
is also capable of projecting complex conceptual graphs into each other, but this is merely an
administrative algorithm that applies simple projections to graphs within contexts; the principle
of maintaining and discarding partial projections is generally the same as that found in
Algorithm 3.4.
Algorithm 3.4 Projection of a Conceptual Graph $u$ Into Another Conceptual Graph $v$: Projection takes two conceptual graphs - a projector $u$ and a projectee $v$ - as arguments, and returns a set of conceptual graph $\{w\}$ that are the projections of $u$ into $v$. The algorithm is based on a node-walking strategy and is recursive; it also requires the selection of a concept $c$ in $u$ and the location of all specialisations of this single concept $c$ in $v$ - the initial elements of $\{w\}$ are copies of this single node.

If $c$ is the only node of $u$, then this precursor stage generates the whole projection; otherwise, $c$, $u$, $v$ and $\{w\}$ are then fed into the following recursive algorithm:

\[
\text{for each non-traversed dyadic relationship } d \text{ of } c \text{ do} \\
\quad \text{traverse a relation } r \text{ to another concept } k \text{ in } u; \\
\quad \text{for each partial projection } w \text{ of } \{w\} \text{ do} \\
\quad \quad \text{find } C, \text{ the projection of } c \text{ in } w; \\
\quad \quad \text{for each dyadic relationship } D \text{ of } C \text{ do} \\
\quad \quad \quad \text{attempt to perform a similar traversal in } w; \\
\quad \quad \quad \text{if a traversal in } D \text{ exists then} \\
\quad \quad \quad \quad \text{if } D \text{ not previously used to extend } w \text{ then} \\
\quad \quad \quad \quad \quad \text{copy } D \text{ (the relation } R \text{ traversed and} \\
\quad \quad \quad \quad \quad \text{the concept } K \text{ reached);} \\
\quad \quad \quad \quad \quad \text{extend } w \text{ with the copy of } D; \\
\quad \quad \quad \quad \text{endif} \\
\quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \text{discard } w; \\
\quad \quad \quad \quad \text{endif} \\
\quad \quad \text{enddo} \\
\quad \text{enddo} \\
\text{enddo}
\]

perform recursion, using $k$ as $c$;

enddo
3.5.1.4 Encapsulation Operations

Given that C-GRASS operates a maximally expanded strategy with regard to its conceptual graphs, the operation of maximal expansion is vitally important. Maximal expansion of a graph \( u \) combines the operations of type and relation expansion, but iterates them until \( u \) is composed entirely of primitive types and relations.

---

**Algorithm 3.5 Maximal Expansion of a Conceptual Graph:** Maximal Expansion takes a conceptual graph \( u \) as argument, and returns a conceptual graph \( v \) that is the maximally expanded form of \( u \):

\[
\text{while } u \text{ contains a node that is non-primitive do}
\]

\[
\text{if a concept } c \text{ of } u \text{ has a non-primitive label } l \text{ then}
\]

\[
\text{make a copy } \beta \text{ of the body } b \text{ of the type definition } l = \lambda x \ b \text{ and maximally join } \beta \text{ to } u \text{ on } c;
\]

\[
\text{else}
\]

\[
\text{if a function node } f \text{ of } u \text{ is not primitive then}
\]

\[
\text{make a copy } \beta \text{ of the body } b \text{ of the function definition } l = \lambda x \ b \text{ and maximally join } \beta \text{ to } u \text{ on } c;
\]

\[
\text{else}
\]

\[
\text{if a relation } r \text{ of } u \text{ has a non-primitive label } l \text{ between concepts } c \text{ and } k \text{ then}
\]

\[
\text{make a copy } \beta \text{ of the body } b \text{ of the relation definition } l = \lambda x, y \ b \text{ and join the genera of } \beta \text{ to } u \text{ on } c \text{ and } k;
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{enddo}
\]
In contrast to the importance of maximal expansion, contraction is of minimal importance, and is only used when a maximally expanded graph needs to be displayed in a conveniently compact form. It is therefore not particularly important in the design of C-GRASS, and its algorithm is omitted for this reason.

3.5.1.5 $\phi_\Delta$

As the fetching of data from secondary storage incurs many overheads, it is obviously in the best interests of a processor using secondary storage to do as much work at the main memory level as possible, and to defer any secondary accesses until the last possible moment. The algorithm for computation of $\phi_\Delta$ in C-GRASS doesn't actually produce formulas of the tuple relational calculus although its activity is equivalent, with production of relational algebra. To do this, it produces three lists:

- the contents list: a list of all the domains that need to be relationally projected from the secondary store in order to answer the query. This list is composed of three sublists:
  - the view list $V$, which corresponds to the free relation scheme of $\phi_\Delta$, and contains attributes that map to the concepts of the query graph;
  - the alias list $A$, which is broadly analogous to the list of range atoms in $\phi_\Delta$, and where each element is a pair $(a, s)$ where $a$ is an alias and $s$ the scheme for which the alias has been generated;
  - the select list $S$, which corresponds to the view domain equalities of $\phi_\Delta$, and where each element consists of an alias and a domain within that alias.

- the joins list $J$: this corresponds to the nonview domain equalities of $\phi_\Delta$, where each element is a pair $(d_1, d_2)$ where $d_1$ and $d_2$ are attributes which join to each other. This list also contain joins resulting from lines of identity across contexts.

- the restrictions list $R$: this corresponds to the referent equalities of $\phi_\Delta$, where each element is a triple $(d, op, v)$ and $d$ is an attribute, $op$ is an range operand, and $v$ is a value.
These lists contain all the information needed to create a view in SQL - or, rather, to create a string of characters that can be interpreted by a dynamic SQL processor. First though, the lists must be created by the use of Algorithm 3.6.

Algorithm 3.6 Formation of $\phi_3$ Lists from a Domain-Extended Conceptual Graph: this algorithm takes a domain-extended conceptual graph $u$ as input, and produces the lists $V, A, S, J$ and $R$ as output.

for each subgraph $g$ of the query graph $u$ do
    form an alias, and add it and its scheme to $A$;
    for each attribute $a$ of $g$
        if the concept mapping to $a$ is not in $V$ then
            add the concept label and its referent to $V$ and $a$ to $S$;
        else
            make an equijoin pair $(a, b)$ of $a$ and the attribute $b$ of the previous appearance in $J$;
        endif
        if the concept mapping to $a$ has a restriction then
            add it to $R$ if it not already there;
        endif
    enddo
enddo
for each concept $c$ of $u$ do
    if $c$ participates in a line of identity with another concept $k$ then
        make an equijoin pair $(a, b)$ of one of the role domain mappings of $c$ and $k$ - attributes $a$ and $b$ respectively - and add it to $J$;
    endif
endo
A view-forming instruction is formed from the lists by Algorithm 3.7.
Algorithm 3.7 Formation of a View from $\phi \Delta$ Lists: this algorithm takes the lists $V$, $A$, $S$, $J$ and $R$ as input, and produces an SQL instruction to build a call as output.

1. Use the attributes $V_1$ ... $V_n$ in $V$ to form the string CREATE VIEW $V$ ($V_1$, ..., $V_n$).
2. Use the elements $S_1$ ... $S_n$ of $S$ to form the string AS SELECT $S_1$, ..., $S_n$.
3. Use the elements $A_1$ ... $A_n$ of $A$ to form the string FROM $A_1.aA_1.s$, ..., $A_n.aA_n.s$.
4. Use the elements $R_1$ ... $R_n$ of $R$ to append the string WHERE $R_1.d$ $op$ $v$ AND ... AND $R_n.d$ $op$ $v$.
5. For each $s \in S$ where $s \notin R$, append the string AND $s_1$ IS NOT NULL AND ... AND $s_n$ IS NOT NULL.
6. Use the elements $J_1$ ... $J_n$ of $J$ to form the string AND $J_1.d$ $1$ = $J_1.d$2 AND ... AND $J_n.d$ $1$ = $J_n.d$2.

The above $\phi \Delta$ algorithm has no problem coping with multiple uses of the same relation scheme. As an example, consider the graph of Figure 3.19(a), which maps to the relation schemes of Figure 3.19(b).

This graph contains a number of identical subgraphs, and so will need to use certain relation schemes more than once.

The solution to this problem is to use aliases - as the same domain can’t be reused in a given scheme without causing confusion, different names (aliases) are given to multiple uses of the same scheme. This idea can be seen if the above example is broken down into its component subgraphs, as in Figure 3.20.
Figure 3.20 Subgraphs of Figure 3.19

Figure 3.20(a) represents a use of the relation scheme R1, and so all role domain mappings share the same alias. This produces the lists in Figure 3.21(a). The next subgraph (Figure 3.20(b)) also uses R1 as a relation scheme, joining on [PERSON: *1]. A new alias T2 is spawned and added to A. As [PERSON: *1] is already in V, it is not added to A or S but instead spawns a join in J (Figure 3.21(b)). The other concepts are both added to V and S.

The next subgraph (Figure 3.20(c)) uses R2 as a relation scheme, joining with previous aliases on [PERSON: *1]. A new alias T3 is therefore spawned, and both [PERSON: *5] and [TALLNESS: *7] are added to V and S, whilst T3 is added to A. Because [PERSON: *1] is a join site, it is added to J but not to S or V (Figure 3.21(c)).
Figure 3.21 Lists of $\phi_A$

(a) \[ V = \{\text{PERSON}_1, \text{CAR}_#80, \text{PLATE}_2\} \]
\[ A = \{(T_1, R_1)\} \]
\[ S = \{T_1.C1, T_1.C2, T_1.C3\} \]
\[ R = \{(T_1.C2, =, #80)\} \]
\[ J = {} \]

(b) \[ V = \{\text{PERSON}_1, \text{CAR}_#80, \text{PLATE}_2, \text{CAR}_#81, \text{PLATE}_3\} \]
\[ A = \{(T_1, R_1), (T_2, R_1)\} \]
\[ S = \{T_1.C1, T_1.C2, T_1.C3, T_2.C2, T_2.C3\} \]
\[ R = \{(T_1.C2, =, #80), (T_2.C2, =, #81)\} \]
\[ J = \{(T_1.C1, T_2.C1)\} \]

(c) \[ V = \{\text{PERSON}_1, \text{CAR}_#80, \text{PLATE}_2, \text{CAR}_#81, \text{PLATE}_3, \text{PERSON}_5, \text{TALLNESS}_7\} \]
\[ A = \{(T_1, R_1), (T_2, R_1), (T_3, R_2)\} \]
\[ R = \{(T_1.C2, =, #80), (T_2.C2, =, #81)\} \]
\[ J = \{(T_1.C1, T_2.C1), (T_1.C1, T_3.C1)\} \]

(d) \[ V = \{\text{PERSON}_1, \text{CAR}_#80, \text{PLATE}_2, \text{CAR}_#81, \text{PLATE}_3, \text{PERSON}_5, \text{TALLNESS}_7, \text{PERSON}_4\} \]
\[ A = \{(T_1, R_1), (T_2, R_1), (T_3, R_2), (T_4, R_3)\} \]
\[ R = \{(T_1.C2, =, #80), (T_2.C2, =, #81)\} \]
\[ J = \{(T_1.C1, T_2.C1), (T_1.C1, T_3.C1), (T_3.C3, T_4.C1)\} \]

(e) \[ V = \{\text{PERSON}_1, \text{CAR}_#80, \text{PLATE}_2, \text{CAR}_#81, \text{PLATE}_3, \text{PERSON}_5, \text{TALLNESS}_7, \text{PERSON}_4, \text{CAR}_6\} \]
\[ A = \{(T_1, R_1), (T_2, R_1), (T_3, R_2), (T_4, R_3), (T_5, R_1)\} \]
\[ R = \{(T_1.C2, =, #80), (T_2.C2, =, #81)\} \]
\[ J = \{(T_1.C1, T_2.C1), (T_1.C1, T_3.C1), (T_3.C3, T_4.C1), (T_4.C2, T_5.C1)\} \]
The next subgraph to be processed is Figure 3.20(d), which maps to the relation scheme R3. The alias T4 is generated and added to A, and also note that [PERSON: *5] has already been used in V, so a join is added to J (Figure 3.21(d)).

Finally, the last subgraph is Figure 3.20(e), which is another use of the scheme R1. [PERSON: *4] appears in V, and so only [CAR: *6] is added to V and S; [PERSON: *4] is added to J (Figure 3.21(e)).

The view-forming instruction for the final set of lists (Figure 3.21(e)) is the SQL instruction (formed by use of Algorithm 3.7) of Figure 3.22. This requires four joins - a binary relational strategy would have required double this number.

Apart from just being used in querying, this mechanism is also quite useful when C-GRASS is trying to work out where to store the data found in incoming assertions - remember that the data is stripped out and stored in relation schemes, and the contents list can be used to do this.

```
CREATE VIEW V
(C1_Person_1, C2_CAR_#80, C3_PLATE_2, C4_CAR_#81, C5_PLATE_3,
C6_Person_5, C7_TALLNESS_7, C8_Person_4, C9_CAR_6)
FROM R1 T1, R1 T2, R2 T3, R3 T4, R1 T5
WHERE T1.C2 = '#80' AND T2.C2 = '#81'
AND T1.C1 IS NOT NULL AND T1.C3 IS NOT NULL AND T2.C3 IS NOT NULL
AND T5.C2 IS NOT NULL AND T1.C1 = T2.C1
```

Figure 3.22 View Created From Figure 3.19 by $\phi_A$
3.5.2 Macro Operations in C-GRASS

Although the algorithms described above could be used in more or less any conceptual graph processor, C-GRASS chooses to hide these basic, graph-level operations from the user by bundling them up in a higher-level, more powerful 'macro' conceptual graph-based language.

In C-GRASS, there are four such basic macro operations - ASSERT, DISPLAY, INQUIRY and QUIT. Of these, ASSERT and DISPLAY operate over all the seven basic structures described earlier; INQUIRY operates solely over application graphs, whilst QUIT has no arguments.

In this section, the more system level aspects of C-GRASS are examined, and a fairly detailed algorithmic treatment of how the macro operations of C-GRASS are performed is given; elaboration of any non-standard requirements arising from database use are also amplified where appropriate. Examples of these operations are also to be found in the user manual for C-GRASS (Appendix C).

3.5.2.1 Application Graph Assertion

The assertion of a conceptual graph into the application graph(s) \( w \) is processed by Algorithm 3.8.

---

**Algorithm 3.8** Assertion of a Conceptual Graph: Assertion takes a conceptual graph \( u \) and the application graph(s) \( w \) as its arguments.

\[
\text{if } u \text{ is uncanonical then } \\
\text{terminate assertion;} \\
\text{else} \\
\text{if } u \text{ contains any nonprimitive and undefined nodes then } \\
\text{terminate assertion;} \\
\text{else}
\]

---

157
maximally expand $u$ and evaluate any functions present;
if evaluation of functions in $u$ leads to inconsistency of $u$
then
    terminate assertion;
else
    test $u$ for theorem proof;
    if $u$ evaluates to TRUE then
        $u$ can be derived from $w$;
    else
        if $u$ evaluates to FALSE then
            $u$ not consistent with $w$;
        else
            check $u$ for cardinal extensions;
            if $u$ does not project into $w$ then
                make a copy $v$ of $u$;
                remove data present in $v$;
                maximally join $v$ to $w$;
                project $u$ into $w$ again;
            endif
            use $\phi_A$ on projections $\pi$ of $u$ into $v$
to form lists;
            use lists to form SQL insertions;
            if a duplicate key error occurs then
                attempt a tuple merge $M$;
                if $M$ fails then
                    terminate assertion;
            endif
        endif
    endif
endif
maintain structurality of $w$;
Algorithm 3.8 is a complex operation, and has many steps that require further explanation, which is now provided.

Firstly: although the algorithm tests $u$ for any nodes that are non-primitive and undefined, what does this actually mean, and what is its significance? The notion of the primitiveness of a type or relation is as follows: if a type label is a direct subtype of a metatype, it is primitive and needs no definition; if a relation label is a direct subrelation of the universal relation, it is primitive and needs no definition. If $u$ contains any undefined non-primitive nodes, the assertion must be rejected; the reason for this is that storage depends upon the ability of the system to expand an assertion into primitive nodes, as seen earlier in the discussion on storage strategies - if such an expansion is not possible, then storage cannot be guaranteed and so graphs containing undefined and non-primitive nodes are therefore rejected.

Secondly, the evaluation of functions may lead to inconsistencies if the result of a functional evaluation disagrees with the assertion - it may be that a function $f$ has a sink concept $c$ that is instantiated, and that the overall evaluation of functions in $u$ may result in an instantiation of $c$ that disagrees with the explicitly asserted instantiation - this is an inconsistency, and causes termination of the assertion.

Thirdly: if $u$ is unknown as a result of theorem proof, $u$ must be tested in order to discover if it needs to be extended as a result of cardinality and membership considerations - these are dubbed cardinal extensions.
In order to do this, each concept $c$ in $u$ is examined and the relations $\{r\}$ it participates in are determined; for each $r \in \{r\}$, if $r$ is known to participate in mandatory relationships, then a cardinal extension may be required. For example, consider the graphs of Figure 3.23. If the relation (POSS) of Figure 3.23(a) is one-to-one and has optional membership on both types, and the relation (POSS) of Figure 3.23(b) is one-to-one and has mandatory membership on PERSON, then if the graph of Figure 3.23(c) is asserted, it can be seen that:

\[\text{[CAR: #1]} \rightarrow \text{(POSS)} \rightarrow \text{[PERSON: #2]}.\]

Figure 3.24 A Graph Extended by Cardinality and Membership
• CAR appears in but one of the cardinalities, and is optional in any case - so no further information is mandated.

• PERSON appears in an optional membership in the first record (also no problem), and in a mandatory membership role in the second - which effectively states that if a person exists, then that person must be the possessor of at least / at most one banana.

A new graph is therefore created (Figure 3.24) that is cardinally extended. This extension guarantees that there will be no null values in the relation schemes into which Figure 3.24 will eventually be stored.

Fourthly: if a projection of \( u \) into \( w \) cannot be found, then \( u \) extends the application domain and \( w \) must be extended to take account of this; \( u \) must be added to the application domain \( w \), which must also be modified to propagate \( \mu_r \) mappings into those parts of \( w \) that were previously the copy of \( u \).

Fifthly: if a duplicate key error occurs in the database as a result of any SQL insertion \( i \), the key uniqueness of a relation scheme has been violated. This means one of two things:

• the assertion is deficient and must be rejected.
• the tuple that is being inserted by \( i \) contains information that can ‘overwrite’ the existing tuple causing the error.

In order to discover which of these possibilities is correct, a tuple merge of the two tuples \( i \) and \( j \) needs to be performed, giving a third tuple \( k \). For each attribute \( a_1, ..., a_n \) in \( i \) and \( j \):

• if \( a_x \) of \( i \) is instantiated and \( a_x \) of \( j \) is not, then \( a_x \) of \( k \) is made the same as \( a_x \) of \( i \), where \( 1 \leq x \leq n \).
• if \( a_x \) of \( j \) is instantiated and \( a_x \) of \( i \) is not, then \( a_x \) of \( k \) is made the same as \( a_x \) of \( j \), where \( 1 \leq x \leq n \).
• if neither \( a_x \) of \( j \) or \( a_x \) of \( i \) is instantiated, then \( a_x \) of \( k \) is \( a_x \) of \( i \) (although it could equally well have been \( a_x \) of \( k \)), where \( 1 \leq x \leq n \).
if \( a_x \) of \( i \) is individuated and \( a_x \) of \( j \) is also instantiated - where \( 1 \leq x \leq n \) - then the tuple merge of \( i \) and \( j \) fails and \( k \) cannot be formed.

If \( k \) cannot be formed, the assertion is deficient; the user is informed of this fact and the assertion is terminated; if \( k \) can be formed, it replaces \( j \) in the relation scheme of which \( j \) was a tuple.

Sixthly (and lastly): the requirement to test the structurality of \( w \) with respect to \( u \) is a consequence of the separation of data and semantics. For example, if the graph of Figure 3.25(a) is \( w \), and that some tuples in the underlying relation schemes allow Figure 3.25(b) to be instantiated. If the graph of Figure 3.25(c) is asserted, it will quite happily project into \( w \) because there are underlying structures prepared to store it; however, merely storing it in this fashion loses the fact that a cycle has come into being, and this is not obvious from \( w \) as it stands. C-GRASS therefore tests to see if the assertion of \( u \) modifies the structurality of \( w \); the above
example would result in \( w \) becoming the graph of Figure 3.25(d), and C-GRASS would modify \( w \) to achieve this structurality.

### 3.5.2.2 Canonical Model Assertion

The assertion of graphs into the canonical model \( c \) is processed by Algorithm 3.9.

---

**Algorithm 3.9** *Assertion of a Graph into the Canonical Model*: the algorithm for the addition of a conceptual graph \( u \) to the canonical model \( c \) is as follows.

1. **if** \( u \) contains instantiated referents **then**
   terminate assertion;
2. **else**
   1. **if** \( u \) duplicates any part of \( c \) **then**
      terminate assertion;
   2. **else**
      1. copy \( c \) to \( k \) and maximally join \( u \) to \( k \);
      eliminate redundant dyads in \( k \);
      2. **if** \( k \) violates the property of canonical subtyping **then**
         terminate assertion;
   3. **else**
      1. **if** \( k \) contains non-primitive undefined nodes **then**
         terminate assertion;
      2. **else**
         1. copy \( k \) to \( \kappa \) and maximally expand \( \kappa \);
         test \( \kappa \) for modifications to cardinality and membership information;
         **if** additions / modifications detected **then**
            perform schema restructuring;
As with Algorithm 3.8, this algorithm needs some further explanation of particular points. Firstly, if any concepts of \( u \) contain instantiated referents, then \( u \) is unacceptable as data is not allowed in the canonical model (as it forms an upper and lower bound on what is plausible in the application domain, it must be completely general). Secondly, if the graph \( u \) duplicates any part of \( c \), or conflicts with any part of \( c \), then it is unacceptable. For example, consider the graphs of Figure 3.26. If \( c \) contained Figure 3.26(a), and an assertion tried to add the graph of Figure 3.26(b) to it, the addition would be permissible because it doesn't duplicate any existing information in \( c \).

**Figure 3.26 Adding Information to the Canonical Model**

(a)  
(b)  
(c)
If an assertion tried to add the graph of Figure 3.26(c) to c, this graph would not be permissible because it is a specialisation of a part of c, and so duplicates something already known about. If the enclosure is odd, the procedure is more or less the other way around: the second addition would be permissible - because it forms a lower bound than is present in c - whilst the first addition would not be permissible.

Thirdly, c may not contain any redundant dyads, and so these must be eliminated during the course of an assertion.

For example, the graph of Figure 3.27(a) contains a redundancy in that the GENDER link between the PERSON and MALENESS concepts is just a specialisation of the more general link between the ANIMATE and MALENESS concepts. Therefore, that relation would be eliminated - Figure 3.27(a) would become the disconnected graphs of Figure 3.27(b).

\[\text{[MALENESS]} - \]
\[(GENDER)\leftarrow [\text{ANIMATE}]\]
\[(GENDER)\leftarrow [\text{PERSON}] - \]
\[(GENDER)\rightarrow [\text{FEMALENESS}],.\]

(a)

\[\text{[ANIMATE]}\rightarrow (\text{GENDER})\rightarrow [\text{MALENESS}].\]
\[\text{[PERSON]}\rightarrow (\text{GENDER})\rightarrow [\text{FEMALENESS}].\]

(b)

Figure 3.27 Graph Disconnection in the Canonical Model
Fourthly: the canonical model must be checked to see if canonical subtyping is violated - if the relations in the oddly-enclosed part of the canonical model \( c \) have conceptual attachments that are subtypes of the corresponding relations in the evenly-enclosed part of \( c \), then canonical subtyping has occurred. Generally, if a relation in the evenly-enclosed context of \( c \) has a match in its oddly-enclosed context, its conceptual attachments are tested: if a non-subtype is found, addition of \( u \) to \( c \) would cause inconsistencies. For example, if \( c \) contained the graphs of Figure 3.28 as a result of adding \( u \), \( k \) would be acceptable as the oddly-enclosed [CAR] has the same label as the evenly-enclosed [CAR], and the oddly-enclosed [PERSON-SANS-LICENCE] has a label that is a subtype of the evenly-enclosed [PERSON].

Subtyping assures that both the positive and negative contexts of the canonical model agree with each other. If subtyping within the join of \( c \) and \( u \) is violated, warn the user that \( u \) is incompatible with \( c \), and terminate the assertion.

### 3.5.2.3 Lattice Assertion

The assertion of a lattice adds an arc between two nodes in that lattice, and is processed by Algorithm 3.10.

If the arc \( \alpha \) between a supemode \( sup \) and a subnode \( sub \) is to be added to the type lattice \( T \), then \( sup \) and \( sub \) are simple strings; if the arc is to be added to the relation lattice \( R \), then \( sup \) and \( sub \) are simple strings enclosed by parentheses (for example, (SPOUSE)).
Algorithm 3.10 Assertion of an Arc into a Lattice: the assertion of an arc $\alpha$ is between the nodes $sup$ and $sub$, into a lattice $L$, is as follows.

\[
\begin{align*}
\text{if } sup \text{ is not a node of } L \text{ then} \\
&\quad \text{add } sup \text{ to } L; \\
&\quad \text{if } sup \text{ is non-primitive and undefined then} \\
&\quad \quad \text{user may assert definition for } sup; \\
&\quad \text{endif} \\
&\text{endif} \\
\text{if } sub \text{ is not a node of } L \text{ then} \\
&\quad \text{add } sub \text{ to } L; \\
&\quad \text{if } \alpha \text{ is an Aristotelian arc and } L \text{ is the relation lattice then} \\
&\quad \quad \text{cast } \alpha \text{ from an Aristotelian arc to a non-Aristotelian arc;} \\
&\quad \text{endif} \\
&\text{endif} \\
\text{if a arc } \beta \text{ already exists (directly or otherwise) between } sup \text{ and } sub \text{ then} \\
&\quad \text{if } \alpha \text{ is redundant - no need to add to } L; \\
&\text{else} \\
&\quad \text{test } sub \text{ for primitiveness;} \\
&\quad \text{if } sub \text{ is non-primitive and undefined then} \\
&\quad \quad \text{user may assert definition for } sub; \\
&\quad \text{endif} \\
&\quad \text{add } \alpha \text{ to } L; \\
&\quad \text{if } sub \text{ primitive but is non-primitive after the addition of } \alpha \text{ then} \\
&\quad \quad \text{get definition for } sub \text{ from the user;} \\
&\quad \text{endif} \\
&\text{endif}
\end{align*}
\]
As with the previous algorithms, a number of points need to be commented on here. Firstly, if \( sup \) needs to be added to \( L \), it is added as either a subnode of the universal relation (if \( L \) is the relation lattice) or as a subnode of a metatype (if \( L \) is the type lattice); in the latter case, it is necessary to prompt the user for this information, and the user may abort the assertion here).

Secondly: if it had been noted before the addition of \( \alpha \) that \( sub \) was a primitive node, it must now be tested to see if it still is, and this is the final step of lattice assertion. Consider these two arcs in the relation lattice:

\[
(UNIV) > (R1) \quad (UNIV) > (R2)
\]

This lattice is satisfactory because both \( (R1) \) and \( (R2) \) are primitive, and neither needs a definition - because they are primitive, they cannot be expanded upon; however, if the arc \( (R2) > (R1) \) is asserted, then the direct arc \( (UNIV) > (R2) \) can be eliminated, which makes \( (R1) \) non-primitive - this cannot exist without a definition because without the definition, the now non-primitive cannot expand out to the most primitive forms, and so the storage strategy is compromised. This is why \( sub \) is be tested to see if it is still primitive - if not, it needs a definition (which may be used in the recasting of storage that comes after this for relations).

### 3.5.2.4 Type, Relation and Function Definition

The assertion of a definition - type, relation, or function - is processed by Algorithm 3.11.

---

**Algorithm 3.11 Assertion of Definitions:** the assertion of a definition comprising a label \( l \), a set of formal parameters \( \{p\} \), and a simple conceptual graph \( d \) is as follows.

```plaintext
if not all \( p \in \{p\} \) correspond to a concept \( c \) in \( d \) then
    terminate assertion;
else
```

---

168
if a definition already exists for \( l \) then

terminate assertion - multiple definitions prohibited;

else

if a function is being defined then

if \( d \) doesn’t conform to function construction rules then

terminate assertion;

else

if \( d \) contains non-primitive / undefined nodes then

terminate assertion;

else

add definition to the application domain \( w \);
expand \( w \) and remap the mappings \( \mu_R \) of \( w \);

endif

endif

else

if \( d \) is uncanonical then

terminate assertion;

else

if a type is being defined then

if no direct arc between \( l \) and the genus of \( d \) exists then

terminate assertion;

endif

endif

endif

if \( d \) contains non-primitive / undefined nodes then

terminate assertion;

else

add definition to the application domain \( w \);
expand \( w \) and remap the mappings \( \mu_R \) of \( w \);
Once again, a comment on this algorithm. If $d$ is being proposed as the body of a function definition, functions are rigidly restricted in the shape and attachments that they can make, and so $d$ is tested to make sure that these restrictions are met - the body of a function definition must be composed purely from numeric typed concepts and functions, and any numeric concept cannot be the result of more than one function - if they are not, then the assertion is terminated.

3.5.2.5 Inquiry

Inquiry takes a conceptual graph $u$ as input, returning either a truth value or a set of instantiations (the interpretation of $u$ over the application graphs $w$). They are processed by Algorithm 3.12:

Algorithm 3.12 Inquiry: the algorithm for testing the application graph $w$ for a query graph $u$ is as follows.

if, after removal of double negations, $u$ still contains nested contexts, then
  test $u$ for theorem proof;
if $u$ evaluates to TRUE then
  $u$ is TRUE;
else
  if $u$ evaluates to FALSE then
    $u$ is FALSE;
else
    \( u \) is UNKNOWN;
endif
endif
else
if \( u \) is uncanonical then
    terminate inquiry;
else
    maximally expand \( u \);
evaluate any functions in \( u \);
if evaluation causes inconsistency in \( u \) then
    terminate inquiry;
else
    project \( u \) into \( w \), returning projections \( \pi \);
create a view \( V \);
display graphs from \( V \), for any \( \pi \);
endif
endif
endif

Some commentary on this algorithm.

Firstly, 'normal' inquiries only occur on graphs of the sort represented in Figure 3.29(a-c); if \( u \) contains nested contexts (as in Figure 3.29(d)), then it is evaluated by theorem proof. The other three types are evaluated by projection into \( w \).

Secondly: if there are any functions in \( u \), check that any concept comparisons in its concepts do not disagree with their referents (which may have been instantiated by functional evaluation); if there is disagreement, warn the user of this fact and terminate the inquiry.
Thirdly: if any projections are found, they are passed to $\phi_A$ and a view $V$ is formed. If any tuples are returned from $V$, they are converted into conceptual graphs and displayed.

The Inquiry operation currently operates only direct queries only - there is no scope for the extraction of implicit query instantiations at present. Also, Inquiry cannot process Figure 3.29(d)-type graphs as, although it can use a theorem prover, such a theorem prover is not yet fully available.

3.5.2.6 Miscellaneous Operation - Display and Quit

These seven operations are simple operations that are used to display particular structural items present in the application domain as a whole, and to leave C-GRASS.

The display of definitions - functions, types and definitions - takes a single label $L$ as sole argument (such as DISPLAY TYPE MAN). If $L$ can be found in the list of the relevant
structure, then it is displayed; otherwise, a message indicates to the user that the label is unknown in the application domain.

The display of explicitly graph-based application information - graphs and the canonical model - have no arguments and simply display the application graphs and canonical model respectively. If either of these structures are empty, the user is informed that this is the case. The display of lattices also has no arguments, displaying both the type and relation lattices.

The Quit operation is also very simple, as it rewrites the binary relational repository store by dumping all the semantic information - which had been read into main memory at the start of the user's session and which might well have been added to since then - present in main memory into it, before terminating the C-GRASS session.

3.5.2.7 Future Functionality

Although C-GRASS has the parser functionality to deal with a Delete operation (which behaves more or less like the Assertion operations in syntactic terms), no development of this area has currently been undertaken.

3.6 Conceptual Graph Issues III: Computation in C-GRASS

Although the relational model has no facility for representing and performing simple computations, a large majority of relational model implementations do offer at least some computational capabilities. Generally speaking, computation in relational database systems can be split into two areas: those operations that operate over attributes of relation schemes, and those operations that operate over tuples of relation schemes.
3.6.1 Aggregate Functions

The first class of operations are generally referred to as *aggregate functions*, and include operations such as SUM, AVERAGE, MAXIMUM, MINIMUM, and COUNT. The most salient fact about such functions is that they operate over a set of values and return a single result - for example, the SUM function could be used to total all sales by a salesman in a month. In a conceptual graph environment, these functions could therefore be seen as operating over the *metaclass* of a type.

This is not related to the metatype of a type (as seen elsewhere), but can instead be considered as a second-order conceptual graph that is formed from *metaproperties* of the type to which it refers. For example, if the type definition for the type PERSON includes information about a person’s age, then average age could be a metaproperty of PERSON.

This inherent second-order nature of aggregate functions makes it rather hard to represent them - in a principled manner, at any rate - in a first-order conceptual graph that is being used as a query graph. Because C-GRASS does not support metaclasses (and so does not have any explicit second-order mechanisms when querying), it does not support aggregate functions.

3.6.2 Arithmetic Functions

The second class of operations are regular arithmetic operations (addition, subtraction, division and multiplication) and those that are based on them (such as square, square root, and so on).

Traditionally, these mechanisms have been performed by using *actors* [Sowa 1976, 1984]; these are special-purpose nodes that can be attached to conceptual graphs and which, given suitable instantiations on its inputs, can compute an output value and use it to instantiate another concept (and which, in turn, may cause other actors to ‘fire’ and compute other values).

For example, consider the actor of Figure 3.30(a). Given instantiation of the input concept (Figure 3.30(b)), the TIMES-TWO actor will compute 50 and instantiate its output concept with it (Figure 3.30(c)).
Whilst the usefulness of a computational element is not in dispute, the method in which it is performed is. Actors are essentially a special kind of type - for example, the above TIMES-TWO actor can be defined as a two-argument (one input, one output) λ-abstraction (Figure 3.30(d)), that adds the input argument to itself and outputs the result.

3.6.3 An Argument for the Rejection of Actor Notation

If actors are a special kind of type, why the different representation? Why not just use special-purpose types to perform computation instead? There are two advantages in such an approach:

- existing syntactic structures can be used - there is no need for a diamond notation.
- the order of arguments in many operations is significant - it is much easier to represent this ordering in a graph form than in an actor form, where some form of arc ordering would probably be required.
C-GRASS supports a computational system based on the use of special-purpose types instead of actors, offering a dedicated \textit{function} definition system that is related to the more usual definitional mechanisms of types and relations. There are four primitive functions - \texttt{ADD}, \texttt{DIVIDE}, \texttt{SUBTRACT} and \texttt{MULTIPLY} - which are 'hard wired' into C-GRASS; any attempt to remove these primitives will be resisted by C-GRASS. These primitive functions can be composed to produce more complex functions - for example, the function \texttt{TIMES-TWO} can be represented by the \( \lambda \)-abstraction of Figure 3.31.

All primitive functions have two \texttt{ARGument} relationships and a \texttt{ReSuLT} relationship; each of these conceptual relationships is linked to a concept of numeric metaclass. The arguments of the function are as would be expected, following argument order:

\[
\lambda x \ [\text{ADDITION}] - \\
(Arg1) \leftarrow [\text{NUMBER: } *x] \\
(Arg2) \leftarrow [\text{NUMBER: } *x] \\
(Rslt) \rightarrow [\text{NUMBER: } *y]
\]

Figure 3.31 \textit{An Addition Function}

Functions can also fire in more useful ways than actors: if the two inputs of a primitive function are instantiated, then a result can be computed - just like actors; but if the result and either of the input arguments is instantiated, then it can 'backfire' and provide the other argument.

\textbf{3.6.4 Limitations of Functions}

Functions are limited in the following ways:
apart from the three conceptual relationships outlined above, no other arguments may be possessed by a function. The function's arguments, on the other hand, may link to whatever they want - with the sole constringtion that a number may not act as a target concept for more than one function (thus, it may not be pointed to by more than one RSLT relationship that is participating in functional use).

- as with other definitions in C-GRASS, recursion within definitions is not permitted; this is because only one definition is allowed for each function. However, this does not - as is the case for other definitional types - prevent the use of a rule graph to model functional recursion.

- although arithmetic functions generally turn up in queries in relational database systems, functions aren't explicitly allowed in query graphs in C-GRASS - although the application of maximal expansion during query processing may well cause function nodes to appear (because they may appear in definitions).

3.6.5 Ranges and Comparisons in C-GRASS

In an earlier chapter, a number of extensions to conceptual graph theory were described that were concerned with comparisons of a particular attribute of a relation scheme against a constant, and with comparisons of one attribute against another. The first comparison was labelled as a concept comparison, and the second as a comparison dyad. Whilst a concept comparison such as PAY<5000 can, in theory, be considered as a different type from PAY, the appearance of PAY<5000 in the type lattice is not required in order to be able to use it; rather, it is considered to be those instances of PAY where the referent value is less than 5000. In query terms, such range restrictions act as a further constraint on what is retrieved - for [PAY<5000], it would retrieve concepts like [PAY: 4500] and [PAY: 3278], but not concepts like [PAY: 5001]. As might be expected - given that these comparisons were developed as a method of modelling certain constructions in relational manipulation systems - concept comparisons are only supported in C-GRASS for query graph use at present. However, other situations can be anticipated where they may be of use:
• the deletion of graphs - the graph:

[ DONATION ] -

(QTY)→[ MONEY-IN-POUNDS <1000 ]

(AGNT)→[ PERSON ]

(YEAR)→[ YEAR: 1995 ].

could be used to delete a set of graphs, specifically all those dyads featuring people who

• the assertion of graphs - for storage of the graph:

[ PERSON: #8008 ]←( EARNER )←[ CASH >10000 ]

which says that PERSON #8008 earns over £10,000 - although it isn’t precisely known

• the assertion of definitions - for the type definition:

Type Poor_Student (x) is

[ STUDENT: *x ]←( POSS )←[ MONEY-IN-POUNDS <5550 ].

where the amount - less than £5500 - possessed by that student is a necessary and

These uses of the concept comparison may be developed in the near future.

At present, C-GRASS does not support comparison dyads; however, it is believed that they

would not be difficult to implement in a later version of the system.
3.7 Conceptual Graph Issues IV: Constraints in C-GRASS

The design of the structures and the operations that were conceived in the theoretical treatment of conceptual graphs and the relational model has now been covered. It now remains to consider the design of the only remaining theoretical development - that of constraint mechanisms.

Ensuring consistency of information in a conceptual graph processor can be considered as a sequence of four steps:

- canonical testing.
- lookup of explicitly asserted information.
- deductive proof (simple chaining).
- full proof.

The first restricts system input to plausible graphs and is performed by a canonical test; the second looks for obvious constraint violations (such as something being both true and false at the same time) in the extensional database and is performed by projection; the third and fourth test system input against the intensional database, and are performed by a theorem prover.

3.7.2 Canonical Constraints in C-GRASS

The most obvious and important constraint mechanism available in C-GRASS is canonicity. In this system, the canonical model corresponds to the set of graphs introduced by insight in [Sowa 1976, 1984], from which other graphs can be formed. However, it differs from that concept of canonical graphs in that the canonical model consists of a true set and a false set. Within the true set, canonical graphs are joined maximally, and no depth greater than one is permitted within the false set. The canonical model is similar to the canonical basis described in [Sowa 1984] in that all type labels must be in the type lattice, and the number of graphs in the model must be finite. However, this assumption can be extended to include the conditions that all relation labels must be in the relation lattice (an obvious extension), and modify it slightly so that it does not allow graphs that have instantiated concepts. Any instantiated concept can be generated by
the application of restriction to an uninstantiated concept (or, alternatively, any graph that is a projection of a canonical graph must be canonical)\(^9\). This last point is important, and allows the canonical model to be seen as upper and lower bounds on canonicity - those graphs which are specialisations of the true set (via projection) may be canonical; they are certain to be canonical if they are also generalisations of the false set (via a generalisation operation or antiprojection).

Algorithm 3.13 is used for the testing of canonicity in C-GRASS.

---

**Algorithm 3.13 Testing of a Conceptual Graph Against the Canonical Model:**

Given a graph \(u\) and the canonical model \(v\) as input, a Boolean value is returned.

```plaintext
for each context \(\gamma\) in \(u\) do
    for each graph \(\gamma\) in \(\gamma\) do
        for each concept \(c\) in \(\gamma\) do
            if \(c\) has no relational attachments then
                \(c\) is canonical;
            else
                create a set of dyad graphs \(\{d\}\);
                for each dyad graph \(d\) do
                    project \(d\) into the even context of \(v\);
                    enddo
                if all projections are successful then
                    create a set of dyad graphs \(\{d\}\);
                    for each dyad graph \(d\) do
                        project \(d\) into the odd contexts of \(v\);
                    enddo
                enddo
```

---

\(^9\) As a matter of interest, the canonical formation rule of restriction is itself a special case of the Peirce rule of insertion [Heaton 1994].
if all projections unsuccessful then
  all of \( \{d\} \) are canonical;
else
  part of \( \{d\} \) is uncanonical;
endif
else
  part of \( \{d\} \) is uncanonical;
endif
enddo
enddo

The Boolean value returned is TRUE (if \( u \) is canonical) or FALSE (if \( u \) is uncanonical), and a dyad graph is a graph formed from a single dyad in which \( c \) participates. If all dyads of all concepts in \( u \) are canonical, then \( u \) is canonical.

If circumstances arise in which some part of \( u \) is found to be uncanonical, there is no point in testing the rest of \( u \), and so the test is terminated.

3.7.2 Other Constraints

Apart from the constraints imposed by cardinality and membership information - which have just been described, and which are not visible to the user - C-GRASS also has the nucleus of a theorem prover that is based on the \( \beta \)-rules of Peirce\(^{10} \). Although theorem proof in C-GRASS is

\(^{10}\) although the theorem-proving approach adopted by C-GRASS is the reductionist system described by [Heaton 1994], not the constructionist system described by [Sowa 1984]; the main reason for this decision is that the reductionist system is guaranteed to terminate, whereas the constructionist system cannot guarantee this whilst remaining complete and computationally tractable (as has been observed by [Fargues et al 1986]). The reductionist approach also requires fewer of the logical operations described by Peirce, and so is unquestionably more efficient than the constructionist approach.
not yet fully functional at the time of writing, all Peirce operations except the operation of Insertion have been implemented.

Because theorem proof cannot be guaranteed to be complete without this operation, C-GRASS does not as yet have access to any mechanism of theorem proof (although mechanisms such as cardinality still make the constraint mechanisms of C-GRASS more powerful than in relational databases, for instance).

3.8 Summary

The theory developed in Chapter 2 would be useless if it could not be implemented, and this chapter has detailed the design behind the development of the C-GRASS processor that has been implemented by the author. It only remains to consider some of the historical ways in which C-GRASS was developed, and how well C-GRASS functions.
CHAPTER 4
Notes on the Implementation
of C-GRASS

In the preceding chapters, a theoretical basis for mapping conceptual graph theory to the relational model has been described, as well as the design of such a mapping and the general-purpose conceptual graph processor that would manage it. The next logical step is obviously to outline how all of this effort can be mechanically implemented to produce some usable tool or processor.

4.1 Chapter Objectives

The aim of this chapter is to discuss, at a fairly detailed level, how C-GRASS is implemented. It will consider what structures are used by C-GRASS to internally represent the application domain, as well as considering the organisation of the various major components of the system and commenting upon how they developed and what problems were faced in their implementation.

4.2 Overall System Operation

A typical C-GRASS session starts with the system reading in data that the user entered in previous sessions, and finishes with that data (along with anything added to the application domain in the current session) being written back to storage. Within these two ‘bookend’ activities, the user is free to display, interrogate, and add to the information in the application domain.
4.2.1 Evolution of Storage System

As has been indicated in the preceding design chapter, the earliest implementations of C-GRASS were based on a binary relational storage strategy. For reasons that have already been seen, it gradually became obvious to us that although such a strategy was easy to implement, it was both unoriginal and computationally inefficient. Most of the work described in [Bowen and Kocura 1993] (reprinted in this thesis as Appendix D) was therefore abandoned over the course of the following year in favour of a more adaptive approach to storage, and this conceptual transition can be seen in [Bowen and Kocura 1994] (reprinted in this thesis as Appendix E).

4.2.1.1 Move From Binary Relational to N-ary Relational Strategy

Although abandoned, the work on binary relational storage was not completely wasted; as has been previously mentioned, an N-ary relational strategy has no facility for the representation of metainformation, such as hierarchy and canonicity. The work on the binary relational approach therefore developed into a repository for the storage of this metainformation.

The earliest binary relational structures used had been extended forms of triple stores, in that they also possessed attributes for:

- the storage of referents.
- the differentiation of different graph structures (definitions, canonical model, etc) within the store.
- the relationship of graphs to other graphs (generally for the representation of contextual domination).
- the depth of a context, and what graphs were inside it.

These structures - the Graph Store and the Context Store - had the structure shown in Figure 4.1.
In Figure 4.1, the Graph Store is organised to store extended graph dyads - it has two concepts with label and referent (FLABEL, FMARK, TLABEL, and TMARK), connected together by a predicate relationship (PRED). Each dyad belongs to a particular graph recognised by the system (GRAPH_ID), and appears in a particular context (CONTEXT_ID). Each dyad is also part of a particular graph structure (STRUCTURE); if that structure is some form of definition, it will also belong to some identifiable definition (LABEL), and the dyad may refer to a concept that is a genus of the definition (FGENUS, TGENUS). The contextual information of the Graph Store is connected together by the Context Store, which features the depth of enclosure (DEPTH) of each context in the Graph Store (CONTEXT_ID), along with which other contexts it dominates (DOMINATING_ID) and the graph structure which this information refers to (STRUCTURE) - if it is a definition, it once again belongs to an identifiable definition (LABEL).
Although such structures turned out to be inefficient from the point of view of storing both data and structure, they turned out to be quite efficient at storing just structure - with the data that they previously stored being separated out into other relation schemes, from which retrieval is also more efficient. With very little alteration, then - in fact, the only major structural change has been the addition of attributes to deal with mappings (FTAB, FDOM, TTAB, TDOM) to the relation names and attributes of these data-storing relation schemes (Figure 4.2) - the two stores have been converted to the storage of structures only.

As mentioned earlier, these repository schemes are primarily used in two circumstances:

- when the user initiates a C-GRASS session; this activates a precursor routine that reads all repository data and converts it into main memory structures that C-GRASS can use.
- when the user terminates a C-GRASS session; this activates a set of write routines that convert all the main memory structures of C-GRASS back into tuples in these schemes.

---

**Figure 4.2 Repository Schemes of C-GRASS**

- (a) Context Store

- (b) Graph Store
The use of such main-memory structures - instead of directly accessing the repository schemes, as used to happen in early system implementations - is also much quicker when processing user requests.

4.2.1.2 Development of Domain Mappings ($\mu_R$)

Whilst a binary relational storage strategy was used by C-GRASS, a formal mapping to the relational model was not really necessary - as it did not really use a system of storage and retrieval that was based on the relational algebra or an equivalent language. With the move away from binary relational storage, the requirement for a formal correspondence became necessary.

The first requirement was the development of a mapping between the structure-storing repository stores (and, by extension, the main memory structures derived from them that would be used by C-GRASS) and the repository schemes that were concerned with the storage of pure data. It was at this stage that the use of conceptual graphs as a system of managing the hyperedges of a hypergraph was conceived, and so a system for corresponding existing data-storing relation schemes to hyperedges was developed - the domain mappings $\mu_R$ and $\mu_C$ of the extended conceptual graph theory.

4.2.1.3 Development of Adaptive Strategy

The abandonment of a binary relational strategy for storage also meant that a system for maintaining the underlying N-ary relation schemes now had to be developed. Previously, a very small number of special-purpose relation schemes had held everything - both information and metainformation - in the form of conceptual graph dyads; now, data and structures were separated. A consequence of this separation was that changes in the application domain might now have an adverse effect on the efficiency of storage and retrieval to and from the data-storing relation schemes - but only if those schemes were static.

In a dynamic application domain, it was therefore necessary for C-GRASS to have the ability to modify those data-storing relation schemes in line with changes to the domain. As has been
previously noted, this is a problem of normalisation, and keeping the data-storing relation schemes in at least the third normal form is necessary if storage and retrieval problems are to be avoided. However, producing a time-efficient automation of this requirement was not easy. Perhaps the best approach adopted was one of the earliest, which selectively modified only those relation schemes for which modification was mandated by changes in the application domain. Such an approach would have been the most desirable, but its complexity of design made it too intractable for development within this research.

Therefore, an alternative - if less time-efficient - approach was implemented, whereby all relation schemes are potentially modifiable upon a change in application domain. Although this ‘universal modification’ was less preferable to selective modification, it was a more tractable objective - although it is more of a scattergun approach that catches the modifications in its blast, rather than a scalpel which zones in on the change efficiently. This is the approach upon which Algorithm 3.1 of Chapter 3 is based.

4.2.1.4 Metatyping and the Storage of Generic Referents

Moving on from the theoretical evolution of storage structure, the implementation issues behind what was actually stored in those structures can now be dealt with. Initially, the only data stored by C-GRASS were either individual markers or generic placeholders for such markers. Both of these markers can be represented as strings - for example, “#8008” and “*12345” - and so can be stored within columns of a relation scheme that have CHAR as a data type (in the case of C-GRASS, the default and maximum size of such columns is 25 characters.

Whilst this arrangement worked well, it was not adequate to cope with some of the non-individual marker classes introduced by metatyping. As may be recalled, three metatypes - string, individual and numeric - are used in C-GRASS. The string type behaved in a broadly similar way to individuals - both its instantiations and generic place holders can be represented as strings, and thus stored in a column of a relation scheme with CHAR as a data type.
However, a problem arose with the numeric type. Although a generic string could be treated in the same way as a generic individual marker, numbers could not be. The reason behind this problem lay in the fact that the processing of numbers was required - to be able to perform range comparison and similar operations over them. In order to achieve this processing, it was necessary to store numeric markers in columns of relation schemes that had NUMERIC as a data type, and this precluded the usual treatment of generic markers for the numeric type - a string, such as "*6667", could not be used to represent an unknown number in secondary storage. In order to counter this problem, generic numbers have been stored within a special range within numeric columns - from the system variable GENERIC_TOP (which has the value -990,000,000) to the value -999,999,999, which is the largest negative number storable in C-GRASS. This range allows the storage of up to 10,000,000 generic numeric markers.

For display purposes, generic numeric markers are translated back into more regular markers (such as *88), and operations which extract tuples from secondary storage purposely ignore this range of numbers unless looking for generic numeric markers. For instance, a projection of the concept [SALARY<10,000] will not retrieve any generic numeric markers, even though they are less than ten thousand in value - the instruction to Oracle contains a condition that, unless the projection is itself generic (such as [SALARY], which retrieves ever salary, generic or otherwise), then numbers less than GENERIC_TOP are to be ignored.

4.2.1.5 Some Intractable Problems Remaining

The move away from special-purpose storage structures with a small number of attributes gives rise to a number of minor problems that are artefacts of the Oracle database management system, and are more-or-less out of the scope of this research to remedy.

Firstly, Oracle has limited indices on tables - therefore, C-GRASS has no 'indexing' beyond the uniqueness constraints that it puts on certain attributes. After all, who can say which attributes should be indexed - as there is no real 'key theory' operable in the theory described in this thesis, beyond that of uniqueness. The small number of attributes in binary relational structures made it easy to index all attributes and combinations of attributes. In N-ary relational structures,
all attributes would ideally be similarly indexed, but this would require a prohibitive amount of
index maintenance. An early attempt to avoid this problem was based on an 'odometer'
approach similar to that seen in [Williams et al 1986], which reindexed the most frequently-
used attributes after a set number of operations.

Second, it is the case that although C-GRASS can detect *tuple duplicates*, Oracle (at least the
version that C-GRASS is implemented in) can't. This will have to wait for a version of Oracle
that can (version 7.0 is reputed to contain a fix for this problem).

Finally, it is a fact that in the binary relational strategy of early C-GRASS, the schemes used
were of a fixed size; in the current regime, the number of attributes that may be in a relation
scheme is theoretically open-ended. However, the Oracle database management system allows a
maximum of 254 attributes in a relation scheme. Although tests of the C-GRASS
implementation have not yet reached such scheme sizes, it is conceivable that they might - but
what happens when this limit is reached is not certain as the extended theory does not recognise
such a limitation, it being nothing more than an artefact of Oracle's implementation. This matter
cannot be ignored, though, and so C-GRASS will need an extension to its design in order to
cope with these limitations.

4.2.2 Parsing of User Input

Once C-GRASS has read in information from its repository structures, the user may interact
with the system. C-GRASS currently possesses a linear form interface and parser that is based
upon an LALR(0) finite state automaton, and this has been the case throughout all but the
system’s very early developmental history.

Initially, the parser of C-GRASS could only cope with simple dyadic graphs\(^1\) and individual
referents, but theoretical developments quickly showed that the grammars upon which the early

---

\(^1\) C-GRASS has not catered for the modelling of relationships that are not dyadic since early in its developmental
history, on the grounds that (a) monadic relationships are seemingly all concerned with higher-order or meta-
logics - such as time, modality and suchlike, all of which are currently outside the remit of C-GRASS; and (b) the vast
majority of triadic and higher-arity relationships can be reduced to dyadic relationships. Some early versions of C-
A BNF Grammar

In order to deal with the syntactic aspect of the theoretical augmentations described, a parser based upon the following BNF grammar was developed:

<table>
<thead>
<tr>
<th>Level</th>
<th>Action</th>
<th>Parse Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accept</td>
<td>$End</td>
</tr>
<tr>
<td>1</td>
<td>Action</td>
<td>ASSERTION Class1 END</td>
</tr>
<tr>
<td></td>
<td>Action</td>
<td>INQUIRY Graph END</td>
</tr>
<tr>
<td></td>
<td>Action</td>
<td>DELETION Class2 END</td>
</tr>
<tr>
<td></td>
<td>Action</td>
<td>DISPLAY Class2 END</td>
</tr>
<tr>
<td></td>
<td>Action</td>
<td>QUIT</td>
</tr>
<tr>
<td>2</td>
<td>Class1</td>
<td>RELATION TypeField LPAR Var2 RPAR Graph</td>
</tr>
<tr>
<td></td>
<td>Class1</td>
<td>TYPE TypeField LPAR STRING RPAR Graph</td>
</tr>
<tr>
<td></td>
<td>Class1</td>
<td>FUNCTION TypeField LPAR STRING RPAR Graph</td>
</tr>
<tr>
<td></td>
<td>Class1</td>
<td>CATALOGUE GraphBody</td>
</tr>
<tr>
<td></td>
<td>Class1</td>
<td>CATALOGUE LPAR GraphBody RPAR</td>
</tr>
<tr>
<td></td>
<td>Class1</td>
<td>GRAPH Graph</td>
</tr>
<tr>
<td>3</td>
<td>Class2</td>
<td>RELATION TypeField</td>
</tr>
<tr>
<td></td>
<td>Class2</td>
<td>TYPE TypeField</td>
</tr>
<tr>
<td></td>
<td>Class2</td>
<td>FUNCTION TypeField</td>
</tr>
<tr>
<td></td>
<td>Class2</td>
<td>CATALOGUE GraphBody</td>
</tr>
<tr>
<td></td>
<td>Class2</td>
<td>CATALOGUE LPAR GraphBody RPAR</td>
</tr>
<tr>
<td></td>
<td>Class2</td>
<td>GRAPH Graph</td>
</tr>
<tr>
<td>4</td>
<td>Graph</td>
<td>GraphBody</td>
</tr>
</tbody>
</table>

GRASS were able to parse higher-arity relationships, on the grounds that the relationship mapped its concepts naturally into a relation scheme with a higher than usual arity, but the development of storage adaptation gradually superseded this approach.
This grammar incorporates the early Sowa grammar, but tidies up some of its more clumsy points. It also extends it with high-level commands such as ASSERT, and with semantic extensions like functions, ranges, and metatyped referents.
4.2.3 Development of Main-Memory Structures

The user's input to C-GRASS is broken down by the parser into a series of tokens, which are then used to convert the linear form into a series of main-memory structures that can be processed by C-GRASS. The most important of these structures are described below.

4.2.3.1 Description of Graph-Building Structures

Complex conceptual graphs - such as application graphs and the canonical model - are built from six major data structures, namely: ContextListElements, Contexts, ConceptListElements, Concepts, RelationListElements, and Relations. In the following description of these structures, only the most salient components are discussed; those components that are basically administrative in nature (used in processing) are italicised.

**ContextListElements** form the backbone of a conceptual graph, and have the following components:

```
Context *PointedAt;
char Used;
int Ident;
ContextListElt *NextInList;
```

Each ContextListElement points at a single Context structure, and forms part of a list for each graph - if that graph has multiple contexts, then the structural analogue of the graph will have multiple ContextListElements.

Unlike ContextListElements, **Contexts** are 'tangible' structures that are visible to the user, and which have the following components:
ConceptListElt *ListOfMembers;
Context *DominatedBy;
ContextListElt *Dominating;
int Depth;
int Processed;

Each Context has a Depth (from zero upwards) and a pointer to a series of simple graph pointers (ListOfMembers). Each Context is also dominated by another Context (except for the zero context, which is not dominated), and may directly dominate multiple other Contexts - for instance, in the graph:

\[ ( (P)(Q) ) \]

the depth zero Context structure directly dominates the depth 1 structure, which in turn directly dominates two depth 2 structures. The Context structure only record direct domination, as indirect domination can be discovered by traversal of such links through multiple Context structures (as in the example). It should be noted that Contexts do not form a list - each context is attached to a single backbone ContextListElement, which links them into a graph.

ConceptListElements are the first simple graph structure, performing a similar 'backbone' service for simple graphs as ContextListElements do for Contexts. They have the following components:

```
Concept *PointedAt;
char Used;
int Ident;
ConceptListElt *NextInList;
```

If a context contains multiple simple graphs, it will have a list of these structures attached to its ListOfMembers component; if a context is empty, the ListOfMembers component points nowhere.
Moving on to simple graph structures, the first of these - the Concept structure - is as expected by conceptual graph theory, and has the following components:

```c
char Label[25];
char Referent[25];
ConceptListElt *LineOfIdentity;
RelationListElt *RelationList;
RangeElt *Ranges;
int GenusFlag;
int Used;
Concept *Next;
```

The Label and Referent components of this structure are as would be expected. Each Concept may also participate in multiple lines of identity; any Concept participating in such lines will possess a list of ConceptListElements (which point to other Concept structures, possibly in different contexts). Each Concept also has at least one relationship in which it participates. This means that at least one RelationListElement appears in each Concept’s RelationList, even those concepts that do not appear to - even single-concept graphs. This is a peculiarity of C-GRASS that arises because even single-concept graphs require conformity domain mappings, and these are stored in RelationListElement structures - hence the need for at least one relation, even if it doesn’t point to anything.

If a Concept is part of a query graph, it may also have some ranges by which the referents returned are restricted. For instance, a concept might have the range restriction:

```
[SALARY<10000]
```

In such a case, a single RangeElt structure carrying the operand and value of the restriction will appear in the Concept’s list of Ranges; there is conceivably no limit to the length of a list of these restrictions. Finally, each Concept also has a GenusFlag - if the structure is the genus of a type definition (or one of the genera of a relation definition), it will carry either the value
XGENUS or YGENUS here, depending on circumstance. Ordinary concepts have the NOTGENUS value here.

Relations are the other obvious simple graph structure, after Concepts. They are as expected by conceptual graph theory, and have the following components:

```c
char Label[25];
Concept *From;
Concept *To;
char Done;
```

These are very simple - they merely have a label, and a pointer to the two Concept structures to which they are attached; they are therefore limited to dyadic connections between Concepts.

Relation structures are connected to Concept structures by RelationListElements - as many Concepts have connections to many Relations, each Concept has a list of pointers in its RelationList that have the following components:

```c
Relation *PointedAt;
char Used;
int Direction;
char TableName[25];
int Domain;
RelationListElt *NextInList;
```

Each structure has one and only one Relation that is PointedAt, and has a particular Direction - they can therefore be considered to be the arcs that link concepts and relations together in a graph. Importantly, each RelationListElement also carries a domain mapping (TableName and Domain) that map the Concept to which it is attached to a particular series of values in secondary storage. As each concept may appear in several places in secondary storage - which
are dependent upon their participation in a particular relationship - it is sensible to link any
given mapping to the relationship upon which it is dependent. For example, consider the graph:

\[
\text{[PERSON: } *a\text{]} - \\
\text{(CHILD)} \rightarrow \text{[PERSON: } *b\text{]} \\
\text{(ATTR)} \rightarrow \text{[NI-NUMBER].}
\]

[PERSON: *a] participates in two relationships and so has two RelationListElements in its
RelationList. The mappings to which each of these structures refer would be different, as the
structure pointing to the (CHILD) relation might point to the 2-ary N-M scheme:

\[
L_1 \text{ (PERSON PERSON)}
\]

whilst the structure pointing to the (ATTR) relation might point to the 2-ary 1-1 scheme:

\[
L_2 \text{ (PERSON NI-NUMBER)}
\]

As mentioned in the description of the Concept structure, each concept requires at least one
RelationListElement in its RelationList. If a Concept participates in no relations, then it has a
single RelationListElt structure in its RelationList; this structure points nowhere and has null
direction, and so is ignored by most system operations - its most important task is to carry a
conformity domain mapping in the TableName and Domain components.

4.2.3.2 Example of Graph-Building Structures

Figure 4.3 gives an example of a complex conceptual graph in its display form, and follows it
(Figure 4.4) with how that graph would be represented in terms of the structures detailed
previously.
4.2.3.3 **Description of Lattice-Building Structures**

Lattices in C-GRASS are constructed from structures that are separate from the graph-building structures just described, being built from *LatticeElements* and *LatticeElementPointers*.

*LatticeElements* are the vertices of a lattice, and have the following components:

- `char Label[25];`
- `LatticeEltPtr *SubLabels;`

Such structures are simple labels, which also have links to other labels through a list of *LatticeElementPointers*. These are analogous to the directed edges of a lattice, and have the following components:

- `LatticeElt *PointedAt;`
- `int ArcType;`
- `LatticeEltPtr *NextInList;`

Each LatticeEltPointer points to a LatticeElement, as well as having a particular ArcType (either Aristotelian or Logical in a type lattice, and Logical in a relation lattice).
Figure 4.4 Internal Representation of Figure 4.3 in C-GRASS
4.2.3.4 Example of Lattice-Building Structures

Figure 4.5 gives an example of a small lattice, and follows it with how the lattice would be represented in terms of the structures detailed previously.

4.2.3.5 Description of Definition-Building Structures

Definitions in C-GRASS are simple conceptual graphs, and so use the Concept, RelationListElement and Relation structures described previously. In addition, they have an additional Defn structure, which has the following components:

```c
char DefName[25];
char XGenus[25];
char YGenus[25];
ContextListElt *Body;
Defn *NextInList;
```

Each definition has a definition name DefName, and may have up to two genera (XGenus and YGenus contain the variable markers of the genera in the Body of the definition although, in a type or function definition, the YGenus is empty). Note that, although there is facility within C-GRASS to represent complex definition bodies structurally, the current system restricts such bodies to single-context graphs of zero depth, containing a simple graph.

4.2.3.6 Structures in Early Versions of C-GRASS

Early versions of C-GRASS possessed very few main-memory structures - certainly nothing as developed as the system just detailed. Instead, graphs and other structures were formed into small versions of a binary relational store and processed by Oracle. This was woefully inefficient, and so was a prime factor in the change from binary relational to N-ary relational storage, in which faster main-memory structures were necessary for processing.
Figure 4.5 A Lattice and its Internal Representation in C-GRASS
4.2.4 Development of Basic Operations

As C-GRASS has evolved over time, the basic operations which it uses have also evolved too. Details are given here of the implementation of some of these operations.

4.2.4.1 Lattice Searching

When C-GRASS utilised a binary relational storage strategy, everything - both data and structure - were held together, as has been seen, in two storage schemes that handled contexts and simple graphs. Lattice information was also held in the simple graph store, and early lattice searches used the CONNECT BY facility of Oracle\(^2\) to derive a table consisting of all subnodes of a specified node. This approach was prohibitively inefficient and was quickly abandoned in favour of lattices held in main memory - perhaps the first break away from purely secondary storage towards main-memory structures.

Perhaps the most important implementational changes have resulted from the decision to use both Aristotelian and Logical arcs, and from the adoption of a maximally expansive strategy - the implementation of lattice assertion needed to be modified to demand a definition in scenarios where a primitive vertex became non-primitive (see section 3.2.8.2 for an explanation of such scenarios).

4.2.4.2 Processing of Canonical Information: ‘Templates’

As canonical information is essentially second-order in nature, early versions of C-GRASS operated an explicitly second-order canonical model *template* that stored all other metadata found in the application domain as well. This semantically rich data dictionary held details on dependency, domain typing information, relational properties, and hierarchical relationships between the types and relations represented therein. The two main second-order types used in the template were [TYPE] and [RELATION]; however, this didn’t give any way of describing

\(^2\) This facility is not standard SQL. Without CONNECT BY, derivation of subtypes would not have been possible.
the dependency information that is used in the efficient decomposition of the template into relational storage structures. Therefore, the [RELATION] metaconcept was extended into the following system for representation of dependency information:

[FD-RELATION] : a functionally determining relation
[MVFD-RELATION] : a multivalued determining relation
[NFD-RELATION] : a non-determining relation

For example, metagraphs used in this version of the canonical model included the graph fragments of Figure 4.6. Each relation could have multiple determinant types and dependant types - all types mapped to relational storage domains, and these dependency relations were useful constraint mechanisms which govern how the domains may combine.

[FD-RELATION: Spouse] -
(DETERMINANT)\rightarrow[TYPE: Man]
(DEPENDANT)\rightarrow[TYPE: Woman]

[MVFD-RELATION: Teaching_Of] -
(DETERMINANT)\rightarrow[TYPE: Time]
(DETERMINANT)\rightarrow[TYPE: University_Room].
(DEPENDANT)\rightarrow[TYPE: University_Course]

NFD-RELATION: Parent] -
(DETERMINANT)\rightarrow[TYPE: Person]
(DEPENDANT)\rightarrow[TYPE: Person]

Figure 4.6 Early Canonical Template 'Metagraphs'
These metagraphs also carried property information—such as whether a relation was reflexive, transitive, and so on, as well as storing cardinality and membership information about the relation.

Although this approach was perfectly feasible, it was abandoned because it added to the information load required of the user, and required the user to make database design decisions if efficient storage was to be procured—for instance, the need to specify cardinality and membership information, as well as to partitioning conceptual relations into dependency classes—this went against one of the original aims of removing the user from the storage design process.

\[
\begin{align*}
( [T:*x]\rightarrow(R)\rightarrow[T:*y]\rightarrow(R)\rightarrow[T:*z] & \quad ( [T:*x]\rightarrow(R)\rightarrow[T:*z] ) \\
(a) \text{ Transitivity} \\
( [T:*x]\rightarrow(R)\rightarrow[T:*x] & \quad ( [T:*y]\rightarrow(R)\rightarrow[T:*x] ) \\
(b) \text{ Symmetry} \\
( [T:*x]\rightarrow(R)\rightarrow[T:*y]\rightarrow(R)\rightarrow[T:*x] & \quad ([T:*y]\rightarrow(=)\rightarrow[T:*x]\rightarrow(=)\rightarrow[T:*y] ) \\
(c) \text{ Antisymmetry} \\
( [T:*x]\rightarrow(R)\rightarrow[T:*x] ) & \quad (d) \text{ Irreflexivity}
\end{align*}
\]

Figure 4.7 Metagraph Properties as Application Graphs
Instead, the properties of the metagraphs used in this approach were dispersed over several other structures: the canonical model became simply a structure that represented the upper and lower bounds of graph plausibility (see Chapter 2), whilst the specification of cardinality and membership became part of both the application graphs and the simplified canonical model, and was automated to remove design responsibility from the user. Property specification was also devolved to the application graphs - the properties shown in Figure 4.7 could be represented in the canonical template system.

4.2.4.3 Projection

In the very early forms of C-GRASS, projection of a graph u into a graph v fragmented both u and v into set of dyads \( \{d\} \) that were held in relation schemes.

For each dyad of the u-scheme, specialisations in the v-scheme were retrieved and merged with the results returned from previously-processed dyads of the u-scheme. In essence, projection was very like the FIND-TRIPLE retrieval strategies of universal triple stores (section 3.2.3 of this thesis), and possessed all the attendant problems of hierarchy and merging described in that section.

Unfortunately for early C-GRASS, this vital operation was prohibitively time-inefficient (see [Bowen and Kocura 1993] - Appendix D - for a full description of the approach being used in this area at the time) - to use the phrase of [Mylopoulos and Brodie 1990] quoted in the Introduction, it really did “nickel and dime the database to death”. The inefficiency of such a vital operation was a problem, and the switch from binary relational to N-ary relational storage partitioned the existing projection operation into the main memory and secondary components that are referred to as projection and \( \Phi_A \) respectively. This separation of responsibility meant that projection could be performed without reference to the database, and retrieval of information from the database was performed in one operation, instead of the many required by earlier operations, and so the current projection in use no longer “nickels and dimes” the database; it differs from the usual operation of [Sowa 1984] in that it is primarily now a flat-graph matching operation (due to the maximally expansive representation strategy adopted under N-ary
relational storage), and in that it has been augmented to handle the referents that appear in C-GRASS as a result of metatyping of concepts.

4.2.4.4 Development of Maximal Join

As with many other operations used in early C-GRASS, maximal join initially operated over relation schemes, not structures. As with projection, [Bowen and Kocura 1993] contains a description of an algorithm used by this early operation. The switch to N-ary relational storage and the widespread use of main-memory structures pushed join use towards the more conventional form seen in [Sowa 1984] and elsewhere; the main adaptation has been to handle metatypes and the referents that result as a spinoff of that theoretical augmentation.

4.2.4.5 Expansion and Contraction

Rather surprisingly, early versions of C-GRASS did not make much use of expansion - perversely, these versions preferred to maximally contract as much information into as specialised a graph as possible, on the grounds that smaller, more specialised graphs were more quickly processed than larger, uncontracted graphs. With the switch to N-ary relational storage, the situation became more-or-less the opposite - expansion became a key operation, with contraction little used (except for display purposes).

Apart from this unusual history, the encapsulation operators were relatively straightforward to implement - the theoretical augmentations described in Chapter 2 have had very little effect upon their actual operation, instead affecting the circumstances in which they operate instead.

4.3 ‘Desirable But Incomplete’: Some Loose Ends

In the previous sections, implementation issues have been dealt with that, whilst they have been substantially modified over time, still contribute to the system’s overall functionality. This
section deals with some important issues that do not currently appear in C-GRASS, because their implementation is desirable and was attempted, but was incomplete.

4.3.1 Parallelism and C-GRASS

Implementations of conceptual graphs at Loughborough University are not limited to C-GRASS, and another research project currently being undertaken is the representation and processing of conceptual graphs by parallel hardware - in this case, transputer arrays.

It was proposed that the special-purpose hardware and software developed by this research project could be interfaced into the C-GRASS processor architecture - such a combined system would have the dual strengths of high-speed main-memory processing of key conceptual graph operations such as projection, whilst being able to store and quickly retrieve large sets of graphs from secondary storage as a result of those parallel operations. Appendix D of this thesis contains some of the notes and programs that were used in this work.

Although a significant amount of work was undertaken investigating the feasibility of interfacing the two systems together, the results of this work were disappointing. The primary reason behind these disappointing results was the difference in data structures between the two systems. C-GRASS requires property extensions to its main-memory structures as a result of its augmentations to conceptual graph theory - most notably to handle the storage of domain mappings, but also to deal with metatyping and operations such as range processing. Although it had developed in parallel with C-GRASS, the parallel processing engine possessed widely divergent data structures and, as it was not concerned with theoretical extension, did not possess the algorithms necessary to handle such extensions.

Although the efficiency of C-GRASS was not materially affected by this parallel interlude, the idea is still rather attractive, and the work done on developing a combined system only foundered because time was not available to bring the communications and data structures between the two systems into line; more dedicated effort along these lines might yet produce useful results.
4.3.2 Proof and C-GRASS

Although C-GRASS does have a theorem-proving system, it is largely untested and does not figure significantly in the current system (although generally useful operations such as Double Negation Removal are currently used). Another research project under development at Loughborough is the CGP (Conceptual Graph Processor), which is fully detailed in [Heaton 1995].

CGP is the only fully-functional theorem prover based on Peirce's β-rules in existence, and it was proposed that C-GRASS and CGP be interfaced together to provide a fully functional deductive database management system based on Peirce logic.

Although this idea was even more attractive than the parallel research, it produced no significant result. As with the parallel implementation, this failure is probably due to differences in data structures; additionally, it is probably true to say that C-GRASS was not ready for such interfacing - this proposal was undertaken at a time when C-GRASS was switching from binary to N-ary relational storage, and so issues of basic storage were more important than theorem proof. Having said that though, the author is of the opinion that some form of interface between the two systems would now be highly tractable - both systems operate correctly over their chosen domains, and a combined system would be desirable to both parties; further research on this matter would be most valuable.

4.4 Summary: Why Implementation is Vital

If the theory described in Chapter 2 is to be of any use, then it must be both amenable to implementation, and also be computationally tractable. In this chapter, some indication of the implementation of C-GRASS has been given, as well as how it led to direct alterations to both theory and design. When looking at other developments in the field of conceptual graphs, it cannot be stressed enough that attempts are made to implement those developments. The implementation of C-GRASS exposed a number of weaknesses and omissions in the original
theoretical correspondence between conceptual graphs and the relational model, and it is
doubtful that these lacunae would have been detected if implementation and testing had not
taken place. In the opinion and experience of the author, the area of conceptual graph theory has
more than its fair share of theorists who are content to develop unworkable theoretical
extensions and make \textit{ex cathedra} statements about the superiority of conceptual graph theory,
without attempting to implement their extensions. Most extensions of conceptual graph theory
are, in the opinion of the author, rather suspect; attempting implementation at least allows one to
base statements on experience.

In contrast, this research has described a number of tractable extensions to conceptual graph
theory, and how they can be worked into the design of a conceptual graph - relational database
hybrid system called C-GRASS, which has also been implemented. Such a hybrid system is
very different to other conceptual graph processors and main-memory based knowledge
engineering environments because it can efficiently process quantities of information that would
cause almost insurmountable inefficiencies in such systems.

With one omission (that of comparison dyads), C-GRASS embodies the theoretical mapping
between the relational model and conceptual graph theory outlined in Chapter 2. This may well
be the case, but does C-GRASS solve the problems of the relational model that were outlined in
the Introduction? The key issue here is that of theorem proof. With it, C-GRASS is a highly-
sophisticated and self-consistent interface layer that converts a relational database management
system into a deductive database management system based on an object-oriented logic, with
the capability to modify storage automatically; without it, C-GRASS is a semantically rich
interface layer that adds object-oriented logical capabilities, improved constraint mechanism,
and an adaptive component to a relational database management system.

Whilst the latter the case - and is certainly worthwhile (any increase in the semantics of an
application domain is worthwhile), the former would certainly be better, and would allow us to
handle most of the problems of the relational model outlined in the Introduction - it is certainly
the case that a graph processor with the full power of Peirce logic (such as \cite{Heaton1995}) is
capable of this:
• conceptual graphs can do recursion and transitive closure, although this is beyond C-GRASS at the moment.

• conceptual graphs have a well-developed system for handling negative information in an open world context, which is preferable to the negation as failure used by the closed world of the relational model. C-GRASS is capable of modelling negative information without resort to the operation of set difference.

• disjunctive information can’t be processed by the relational model, yet conceptual graph theory can handle it (although C-GRASS currently cannot).

• indefinite data can be handled by conceptual graph theory, which permits something to exist yet have no concrete identity by its use of generic markers. C-GRASS has refined this use into unique generic markers that act as a form of null placeholder where it can be determined that one should appear (when determined by cardinality and membership, for instance).

• although conceptual graph theory has the faculty for the processing of higher-order logics (the $\rho$ and $\tau$ operators), no system yet developed can process this logic; C-GRASS is no exception to this rule.

It can be concluded that, although incomplete from the point of view of solving all of the problems of the relational model, the theoretical augmentations to conceptual graph theory described in this thesis remedy some significant problems whilst being amenable to implementation; the next step is to consider their computational tractability, and this is the subject of the next chapter.
In order to be useful, C-GRASS must meet two important criteria:

- it must be correct in its operation - in the sense that all of its functions return the results demanded by the conceptual graph theory of [Sowa 1984], and the theoretical extensions described in Chapter 2.
- it must be reasonably efficient in its operation - in the sense that the running time required by both individual functions and the overall system is fast, and does not drastically increase as the size and complexity of an application domain increases.

5.1 Chapter Objectives

In order to show that C-GRASS can meet these criteria, this chapter has a number of objectives. The first of these is the identification of important functions within C-GRASS that are widely used and so need to be efficient.

Once functions have been identified in this way, they can be tested. For each such test, the following steps are required:

- a description of what the test is to do, and how it will operate.
- a prediction of what the test should return as a correct result, along with a specification of the running time required to perform it.
• an analysis of the result returned, and a comparison with the predictions made. If the analysis is widely divergent from the predictions, further commentary on the reasons behind the divergence will be made.

The next step after consideration of individual functions is to consider the correctness and efficiency of the overall system. Overall correctness is shown by conducting the reader through a sample application domain (familial relationships), showing how the system reacts to the various assertions required in order to build that domain. The overall efficiency is considered in terms of both how long macro operations (such as assertion and inquiry) take to execute; this also takes into account the efficiency of the database component of C-GRASS.

Finally, some speculations about how different application domains might affect the performance of the system are offered: a domain might have a much deeper lattice, or more graphs, or be structurally more complex - or any number of factors such as these. Some pointers upon the analysis of this matter are offered.

5.2 **Identification of Key Functions in C-GRASS**

Which functions are the most important in C-GRASS? If a function is used frequently, then it must be efficient if the system is not to experience performance problems.

Examination of C-GRASS suggests that there are a well-defined group of functions that appear with regularity:

• lattice searching.
• maximal expansion.
• projection.
• maximal join.
• copy.
• posting.
• the $\phi_A$ operation.

These functions have been organised on the basis of frequency of usage; they can also be neatly partitioned into frequent functions that operate solely in main memory (lattice searching, maximal expansion, projection, maximal join, and copy), and important functions that interface to secondary storage at some level (posting and $\phi_A$).

5.2.1 Main-Memory Functions Under Test

As regards the first group of functions, lattice searching has many uses in many routines and is, without argument, the most widely-used function in C-GRASS; because of its near-universality, its correct and efficient operation over a wide range of lattice depths and bushinesses\(^1\) is vital to C-GRASS. Maximal expansion is also a widely-used operation in C-GRASS because of its strategy of operating over maximally expanded conceptual graphs; its correct and efficient operation is necessary if higher-level functions like Inquiry, Assertion and schema restructuring are to operate effectively.

Although lattice search and maximal expansion are the most widely-used functions, there are other functions that, whilst less frequent in use, are equally as important. The prime example of such functions are projection and maximal join. The projection operation is the cornerstone of higher-level functions such as Inquiry and the testing of canonicity, and so needs to operate correctly and efficiently to ensure the efficiency of those other functions. Maximal join is less important than projection, but has a number of important applications in maximal expansion ensure a high frequency of use and its efficient operation is, amongst other things, necessary to the good operation of expansion.

Finally, the copy operation - although relatively simple - is an important operation, as many copies of graphs are taken by various functions in order to preserve the original input.

\(^1\) The number of subnodes possessed by a node is its bushiness.
5.2.2 Interface Functions Under Test

Looking at the second group of functions, the \( \phi_\alpha \) function is fairly frequently used as it is important in both the assertion and retrieval of data into and from secondary storage; its efficiency and correctness of operation is therefore of vital importance to those macro-operations. The function of posting is less frequent - it occurs when a change in the application domain requires a modification to the underlying relational storage schemes - and is a pivotal component of the important task of schema restructuring.

5.3 Testing of Key Functions

The object of this section is to offer an evaluation of the functions that were identified as important in the previous section. For each function, a series of tests will be performed that are designed to show both the correctness of the function in question, as well as some indication of the efficiency (or otherwise) of the function. For each function, the nature of the tests to be performed is described, and also what sort of result from the test would show correctness of function; a prediction of efficiency based on asymptotic evaluation of the function’s algorithm is also offered. The results returned are analysed in the light of the asymptotic prediction made, and a comparison between theory and reality is attempted.

Appendix F contains details of program dumps and the programs used to test C-GRASS at this level.

5.3.1 A Basic Unit of Testing

Many of the functions that are under test are, unsurprisingly, used in the processing of conceptual graphs, and their efficiency is generally related to the size and complexity of the graph(s) being processed.
Throughout the tests that follow, graphs are used that are constructed from a simple base unit of five dyadic conceptual relationships. This basic unit is shown in Figure 5.1, along with some
graphs constructed from such units. Whenever a 10-dyad graph is referred to in the following tests, it refers to a graph patterned after the fashion of Figure 5.1(b); similarly, reference to a 20-dyad graph refers to a graph patterned after the fashion of Figure 5.1(b), and so on.

5.3.2 Lattice Searching

Lattice searching operates over a structure $L$, composed of a set of vertices $V$ and a set of directed edges $E$ between those nodes. The number of vertices and edges is dictated by the depth $d$ of $L$ and the bushiness $b$ of each node $n \in V$, and these are the values that shall be varied in order to test the efficiency of lattice searching.

5.3.2.1 Description of Testing

The set of tests performed vary both $d$ and $b$ of $L$ over a sequence of nine tests where:

1. $d = 2$ and $b = 2$
2. $d = 2$ and $b = 4$
3. $d = 2$ and $b = 6$
4. $d = 4$ and $b = 2$
5. $d = 4$ and $b = 4$
6. $d = 4$ and $b = 6$
7. $d = 6$ and $b = 2$
8. $d = 6$ and $b = 4$
9. $d = 6$ and $b = 6$

Although the depth and bushiness of the lattices generated by these tests may intuitively seem small, this is not so - for example, consider the number of vertices and edges in a lattice $L$ where $d = 6$ and $b = 6$; for each level of $L$, $V$ increases by $d^2$:

$$1 + 6^1 + 6^2 + 6^3 + 6^4 + 6^5 + 6^6 : V = 55,987 \text{ and } E = 55,987$$

Each test uses a test program to generate lattices based on the values of $d$ and $b$ required by the particular test being performed; it then measures the time taken to traverse the generated lattice from the root node to the very last vertex generated (so in the case where $d = 6$ and $b = 6$, the
time taken to test over 55,000 vertices would be measured). As each vertex (excepting the root) always has only one ‘supervertex’, \( V = E \) in all lattices generated.

### 5.3.2.2 Predicted Results

Tree-scanning algorithms traditionally visit each vertex in \( V \) exactly once, and traverse each edge in \( E \) once in the worst case scenario envisaged by looking for the last node in the lattice; The algorithm used by C-GRASS adopts this strategy - if the node just reached from the supervertex is not the target vertex, then recurse and take the vertex just reached as the supervertex in the new recursion.

Regarding correctness, this will simply be shown if the target vertex can be reached from the root. Efficiency depends on \( V \) and \( E \) (which are dictated by increases in \( d \) and \( b \) - as each vertex and edge are visited once apiece, it is predicted that the running time of the lattice searching algorithm should be proportional to \( V + E \).

### 5.3.2.3 Results and Analysis

The nine tests described above produced the results shown in Figure 5.2(a); Figure 5.2(b) is a graphical representation of those results.

An increase in either \( d \) or \( b \) increases both \( V \) and \( E \) in equal proportions. Do the results produced by our tests agree with our prediction of running time? Consider the number of nodes and edges involved in each test:

1. \( d = 2 \) and \( b = 2 \): \( V = 7 + E = 7 \) = 14
2. \( d = 2 \) and \( b = 4 \): \( V = 21 + E = 21 \) = 42
3. \( d = 2 \) and \( b = 6 \): \( V = 43 + E = 43 \) = 86
4. \( d = 4 \) and \( b = 2 \): \( V = 31 + E = 31 \) = 62
5. \( d = 4 \) and \( b = 4 \): \( V = 341 + E = 341 \) = 682
6. \( d = 4 \) and \( b = 6 \): \( V = 1,555 + E = 1,555 \) = 3,110

217
7. $d = 6$ and $b = 2$: $V = 127 + E = 127$ = 254
8. $d = 6$ and $b = 4$: $V = 5,461 + E = 5,461$ = 10,922
9. $d = 6$ and $b = 6$: $V = 55,987 + E = 55,987$ = 111,974

These figures are represented graphically in Figure 5.2.

<table>
<thead>
<tr>
<th>b\d</th>
<th>2</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>14</td>
<td>62</td>
<td>254</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>682</td>
<td>10,922</td>
</tr>
<tr>
<td>6</td>
<td>86</td>
<td>3,110</td>
<td>111,974</td>
</tr>
</tbody>
</table>

(a) Predicted Figures

(b) Graphical Interpretation

Figure 5.2 Predicted Results for Lattice Searching
Table 5.6

<table>
<thead>
<tr>
<th>b/d</th>
<th>2</th>
<th>4</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>0.02</td>
<td>0.26</td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.08</td>
<td>3.15</td>
</tr>
</tbody>
</table>

(a) Test Figures

Figure 5.3 Test Results for Lattice Searching

Figure 5.3 shows the actual results of the tests performed. It can immediately be seen that the predicted and actual graphs have a very similar shape, and so the actual implementation conforms quite closely to the algorithm upon which it is based. It can also be seen that the algorithm (and thus, the implementation) is generally linear with regard to V and E - for instance, test #6 takes around five times longer to perform than test #5 because it has about five times more vertices and edges to traverse.
5.3.3 Maximal Expansion

Maximal expansion operates over a conceptual graph $u$, and expands any non-primitive concepts or relations out until all nodes of $u$ are primitive. The size of $u$ and the number of expansions $\eta$ are the key factors in the efficiency of maximal expansion, and these are the values that shall be varied in order to test that efficiency.

5.3.3.1 Description of Testing

The set of tests performed vary the size of $u$ (in terms of the number of dyads that form the graph) and the number of $\eta$ over a sequence of ten tests where:

1. $u = 10$ and $\eta = 5$
2. $u = 20$ and $\eta = 5$
3. $u = 40$ and $\eta = 5$
4. $u = 80$ and $\eta = 5$
5. $u = 20$ and $\eta = 10$
6. $u = 40$ and $\eta = 10$
7. $u = 80$ and $\eta = 10$
8. $u = 40$ and $\eta = 20$
9. $u = 80$ and $\eta = 20$
10. $u = 80$ and $\eta = 40$

Tests with similar $u$ are identical in all respects but one - the number of concept nodes that can be expanded. Each expansion site is represented in $u$ as the type label C5, which has the definition:

$$C5 = \lambda x \ [C1: *x] \rightarrow (R1) \rightarrow [C4].$$

Such a definition, once expanded into $u$ in place of the C5 label, cannot be expanded further as it is composed from primitive types and a primitive relation. The time taken to measure all expansions - so that $u$ is maximally expanded - is measured.
5.3.3.2 Predicted Results

Efficiency should be dependent upon the number of expansions $\eta$ in the graph $u$ that are performed. Whilst looking for non-expanded concepts and relations, Algorithm 3.5 of Chapter 3 visits each concept $v \in V$ and relation $e \in E$ once each, so the scanning for non-expanded nodes is $V+E$. As the whole of $u$ is scanned until no non-expanded graphs remain, this scanning must take place $\eta$ times.

Based on this algorithm, it is predicted that the running time of the copy algorithm will therefore be proportional to the following formula:

$$\eta \cdot (V+E)$$

For the test series employed, the following is predicted:

1. $u = 10$ and $\eta = 5$: $V+E = 17, \eta(V+E) = 85$
2. $u = 20$ and $\eta = 5$: $V+E = 32, \eta(V+E) = 160$
3. $u = 40$ and $\eta = 5$: $V+E = 62, \eta(V+E) = 310$
4. $u = 80$ and $\eta = 5$: $V+E = 120, \eta(V+E) = 600$
5. $u = 20$ and $\eta = 10$: $V+E = 32, \eta(V+E) = 320$
6. $u = 40$ and $\eta = 10$: $V+E = 62, \eta(V+E) = 620$
7. $u = 80$ and $\eta = 10$: $V+E = 120, \eta(V+E) = 1200$
8. $u = 40$ and $\eta = 20$: $V+E = 62, \eta(V+E) = 1240$
9. $u = 80$ and $\eta = 20$: $V+E = 120, \eta(V+E) = 2400$
10. $u = 80$ and $\eta = 40$: $V+E = 120, \eta(V+E) = 4800$

These figures are represented graphically in Figure 5.4.
### 5.3.3.3 Results and Analysis

The test suite described above produced the results shown in Figure 5.5(a); Figure 5.5(b) is a graphical representation of those results.

From these results, it can once again be seen that the predicted and actual graphs have a very similar shape, indicating that the implementation is a close approximation of the underlying algorithm.
### Test Figures

<table>
<thead>
<tr>
<th>$u \eta$</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.05</td>
<td>××</td>
<td>××</td>
<td>××</td>
</tr>
<tr>
<td>20</td>
<td>0.09</td>
<td>0.18</td>
<td>××</td>
<td>××</td>
</tr>
<tr>
<td>40</td>
<td>0.16</td>
<td>0.33</td>
<td>0.68</td>
<td>××</td>
</tr>
<tr>
<td>80</td>
<td>0.37</td>
<td>0.62</td>
<td>1.26</td>
<td>2.82</td>
</tr>
</tbody>
</table>

(a) Test Figures

(b) Graphical Interpretation

Figure 5.5 Test Results for Maximal Expansion

It can also be seen that the algorithm (and thus, the implementation) is more or less linear - as $\eta$ doubles, the running time taken to compute them also doubles; this is also true for $u$.

### 5.3.4 Projection

Projection in C-GRASS is broadly the same as that found in [Sowa 1984], although the maximal expansion strategy employed by C-GRASS means that it is most frequently found operating as a flat graph match.
In the absence of a lattice, projection operates over two conceptual graphs \( u \) and \( v \) - respectively the projector and the projectee. The sizes of \( u \) and \( v \) are the values that shall be varied in order to test the efficiency of simple projection.

5.3.4.1 Description of Testing

The set of tests performed vary the size of both \( u \) and \( v \) (in terms of the number of dyads that form the graph) over a sequence of fifteen tests where:

1. \( u = 5 \) and \( v = 5 \)
2. \( u = 5 \) and \( v = 10 \)
3. \( u = 5 \) and \( v = 20 \)
4. \( u = 5 \) and \( v = 40 \)
5. \( u = 10 \) and \( v = 10 \)
6. \( u = 10 \) and \( v = 20 \)
7. \( u = 10 \) and \( v = 40 \)
8. \( u = 10 \) and \( v = 80 \)
9. \( u = 10 \) and \( v = 80 \)
10. \( u = 20 \) and \( v = 20 \)
11. \( u = 20 \) and \( v = 40 \)
12. \( u = 20 \) and \( v = 80 \)
13. \( u = 40 \) and \( v = 40 \)
14. \( u = 40 \) and \( v = 80 \)
15. \( u = 80 \) and \( v = 80 \)

Each test uses a test program to generate both \( u \) and \( v \) from repetitions of five-dyad basic units - a 10-dyad graph is composed from two units, a 20-dyad graph from four, and so on (the repetitiveness of each graph’s component units automatically gives rise to multiple projections, which is a useful spinoff of the basic unit approach). Once \( u \) and \( v \) have been formed, the time taken for the projection of \( u \) into \( v \) is measured and displayed.

5.3.4.2 Predicted Results

Correctness of the tests will be illustrated by the number of projections of \( u \) in \( v \):

1. \( u = 5 \) and \( v = 5 \): 1 5-dyad projection of \( u \) in \( v \)
2. \( u = 5 \) and \( v = 10 \): 2 5-dyad projections of \( u \) in \( v \)
3. \( u = 5 \) and \( v = 20 \): 4 5-dyad projections of \( u \) in \( v \)
It is predicted that the efficiency of projection depends upon both the sizes of $u$ and $v$. Looking back at Algorithm 3.4 of Chapter 3, the initial step of projection requires that all concept nodes of $V_v$ are scanned to find matches of a concept node (any node) in $V_u$, producing an initial set of partial projections, $w$, of that node into $V_v$ (this set of projections changes as the graphs are evaluated). For each partial projection in $w$, the algorithm finds the corresponding node of $i$ in $V_v$ and attempts to extend a corresponding edge of $j$ in $E_v$; if it succeeds, then that element of $w$ survives and is extended. This algorithm is difficult to represent mathematically - An accurate representation might be something like:

$$V_v + \left( \sum_{i=1}^{v} (\sum_{j=1}^{u} (\sum_{k=1}^{E_v})) \right)$$

where $v$ represents the vertices of $V_u$ and $\varepsilon$ the edges of $E_u$, and which scans all of $V_v$ first, before iterating through the concepts $v$ and edges $\varepsilon$ of $u$ and trying to extend the elements of $w$ with $E_v$. However, this is a difficult formula to test, and so in this case it is predicted that the running time of projection will be less than - not proportional - to the following formula:
For the test series employed, the following is predicted:

1. \( u = 5 \text{ and } v = 5 \) : 1 5-dyad projection of \( u \) in \( v = 4 + (25\times1) = 29 \)
2. \( u = 5 \text{ and } v = 10 \) : 2 5-dyad projections of \( u = 7 + (50\times2) = 107 \)
3. \( u = 5 \text{ and } v = 20 \) : 4 5-dyad projections of \( u = 12 + (100\times4) = 412 \)
4. $u = 5$ and $v = 40$ : 8 5-dyad projections of $u = 22 + (200\times8) = 1622$
5. $u = 5$ and $v = 80$ : 16 5-dyad projections of $u = 40 + (400\times16) = 6440$
6. $u = 10$ and $v = 10$ : 1 10-dyad projection of $u$ in $v = 7 + (100\times1) = 107$
7. $u = 10$ and $v = 20$ : 2 10-dyad projections of $u$ in $v = 12 + (200\times2) = 412$
8. $u = 10$ and $v = 40$ : 4 10-dyad projections of $u$ in $v = 22 + (400\times4) = 1622$
9. $u = 10$ and $v = 80$ : 8 10-dyad projections of $u$ in $v = 40 + (800\times8) = 6440$
10. $u = 20$ and $v = 20$ : 1 20-dyad projection of $u$ in $v = 12 + (400\times1) = 412$
11. $u = 20$ and $v = 40$ : 2 20-dyad projections of $u$ in $v = 22 + (800\times2) = 1622$
12. $u = 20$ and $v = 80$ : 4 20-dyad projections of $u$ in $v = 40 + (1600\times4) = 6440$
13. $u = 40$ and $v = 40$ : 1 40-dyad projection of $u$ in $v = 22 + (1600\times1) = 1622$
14. $u = 40$ and $v = 80$ : 2 40-dyad projections of $u$ in $v = 40 + (3200\times2) = 6440$
15. $u = 80$ and $v = 80$ : 1 80-dyad projection of $u$ in $v = 40 + (6400\times1) = 6440$

These figures are illustrated graphically in Figure 5.6.

As the number of projections rises as the size of $v$, the projectee, increases, this prediction makes sense.

5.3.4.3 Results and Analysis

The suite of tests described above produced the results shown in Figure 5.7(a), whilst Figure 5.7(b) is a graphical representation of those results.

It can immediately be seen that the predicted and actual graphs are not exactly as similar as has been expected - the rise in the size of $v$ provokes a more gentle rise in running time than was predicted until $u = 80$ and $v = 80$, where the result is more in line with our expectations. Given the lower confidence in our mathematical analysis for this operation, this is perhaps not surprising.
Like projection, maximal join operates over two conceptual graphs $u$ and $v$. The sizes of $u$ and $v$ are important values in maximal join as well as projection, but any evaluation of maximal join must also take into account the size $j$ of the join between $u$ and $v$, and these are the values that shall be varied in order to test the efficiency of maximal join.
5.3.5.1 Description of Testing

The set of tests performed vary the size of both $u$ and $v$ (in terms of the number of dyads that form the graph) and the size of $j$ (again in terms of dyads) over a sequence of twenty-five tests where:

1. $u = 10, v = 10$ and $j = 5$
2. $u = 10, v = 20$ and $j = 5$
3. $u = 10, v = 40$ and $j = 5$
4. $u = 10, v = 80$ and $j = 5$
5. $u = 20, v = 10$ and $j = 5$
6. $u = 20, v = 20$ and $j = 5$
7. $u = 20, v = 40$ and $j = 5$
8. $u = 20, v = 80$ and $j = 5$
9. $u = 40, v = 10$ and $j = 5$
10. $u = 40, v = 20$ and $j = 5$
11. $u = 40, v = 40$ and $j = 5$
12. $u = 40, v = 80$ and $j = 5$
13. $u = 80, v = 10$ and $j = 5$
14. $u = 80, v = 20$ and $j = 5$
15. $u = 80, v = 40$ and $j = 5$
16. $u = 80, v = 80$ and $j = 5$
17. $u = 20, v = 20$ and $j = 10$
18. $u = 20, v = 40$ and $j = 10$
19. $u = 20, v = 80$ and $j = 10$
20. $u = 40, v = 20$ and $j = 10$
21. $u = 40, v = 40$ and $j = 10$
22. $u = 40, v = 80$ and $j = 10$
23. $u = 80, v = 20$ and $j = 10$
24. $u = 80, v = 40$ and $j = 10$
25. $u = 80, v = 80$ and $j = 10$

Like projection, each test uses a test program to generate both $u$ and $v$ from repetitions of five-dyad basic units - a 10-dyad graph is composed from two units, a 20-dyad graph from four, and so on. Once $u$ and $v$ have been formed, the time taken for the maximal join of $u$ to $v$ is measured.

5.3.5.2 Predicted Results

Correctness of the tests will be illustrated by the joined graphs returned by the various tests. A correct result will be a graph that is only joined upon the dyads which are shared (by virtue of referents) by $u$ and $v$. For example, the result returned by the first test, where $u = 10, v = 10$ and $j = 5$, in Figure 5.8.
[C1 : #1] -
(R1)->[C2 : #2] -
(R2)->[C3 : #3] -
(R4)->[C4 : #4] -
(R3)->[C1 : #1] -
(R5)->[C2 : #5] -
(R3)->[C1 : #5] -
(R1)->[C2 : #6] -
(R2)->[C3 : #7] -
(R4)->[C4 : #4],
(R5)->[C4 : #4]....

(a) Graph u

[C1 : #1] -
(R1)->[C2 : #2] -
(R2)->[C3 : #3] -
(R4)->[C4 : #4] -
(R3)->[C1 : #1] -
(R5)->[C2 : #2] -
(R3)->[C1 : #500] -
(R1)->[C2 : #600] -
(R2)->[C3 : #700] -
(R4)->[C4 : #4],
(R5)->[C4 : #4]....

(b) Graph v

[C1 : #1] -
(R1)->[C2 : #2] -
(R2)->[C3 : #3] -
(R4)->[C4 : #4] -
(R3)->[C1 : #1] -
(R5)->[C2 : #2] -
(R3)->[C1 : #5] -
(R1)->[C2 : #5] -
(R2)->[C3 : #7] -
(R4)->[C4 : #4],
(R5)->[C4 : #4],
(R3)->[C1 : #500] -
(R1)->[C2 : #600] -
(R2)->[C3 : #700] -
(R4)->[C4 : #4],
(R5)->[C4 : #4]....

(c) Joined Graph

Figure 5.8 Correct Result for u = 10, v = 10 and j = 5 (Maximal Join)
Efficiency would seem to be dependent on the size of $u$, $v$, and $j$. Looking back at Algorithm 3.3 of Chapter 3 (which deals with the maximal join of simple conceptual graphs), it can be seen that an initial nested loop of $V_u$ and $V_v$ occurs - the task of which is to find point concept matches between $u$ and $v$, and to try to extend them into local joins $j \in J$. The following step then scans each node of $V_u$ and scans $J$ for the largest join site $j$ in which that node appears (if at all).

Based on this algorithm, it is predicted that the running time of maximal join will be proportional to the following formula:

$$(V_u V_j) + V_u \sum_{i}^{k_j} J_i$$

The $\sum$ within this formula scans all concepts $k$ of $j$ against the set of join sites $J$ to make sure that all $k$ are in the largest join possible. In the following tests, $J = 1$, and can be ignored. For the test series employed, the following is predicted:

1. $u = 10$, $v = 10$ and $j = 5$  
   $V_u = 7$, $V_v = 7$, $(V_u V_v) + V_u = 56$
2. $u = 10$, $v = 20$ and $j = 5$  
   $V_u = 7$, $V_v = 12$, $(V_u V_v) + V_u = 91$
3. $u = 10$, $v = 40$ and $j = 5$  
   $V_u = 7$, $V_v = 22$, $(V_u V_v) + V_u = 161$
4. $u = 10$, $v = 80$ and $j = 5$  
   $V_u = 7$, $V_v = 40$, $(V_u V_v) + V_u = 287$
5. $u = 20$, $v = 10$ and $j = 5$  
   $V_u = 12$, $V_v = 7$, $(V_u V_v) + V_u = 96$
6. $u = 20$, $v = 20$ and $j = 5$  
   $V_u = 12$, $V_v = 12$, $(V_u V_v) + V_u = 156$
7. $u = 20$, $v = 40$ and $j = 5$  
   $V_u = 12$, $V_v = 22$, $(V_u V_v) + V_u = 276$
8. $u = 20$, $v = 80$ and $j = 5$  
   $V_u = 12$, $V_v = 40$, $(V_u V_v) + V_u = 492$
9. $u = 40$, $v = 10$ and $j = 5$  
   $V_u = 22$, $V_v = 7$, $(V_u V_v) + V_u = 176$
10. $u = 40$, $v = 20$ and $j = 5$  
    $V_u = 22$, $V_v = 12$, $(V_u V_v) + V_u = 286$
11. $u = 40$, $v = 40$ and $j = 5$  
    $V_u = 22$, $V_v = 22$, $(V_u V_v) + V_u = 506$
12. $u = 40$, $v = 80$ and $j = 5$  
    $V_u = 22$, $V_v = 40$, $(V_u V_v) + V_u = 902$
13. $u = 80$, $v = 10$ and $j = 5$  
    $V_u = 40$, $V_v = 7$, $(V_u V_v) + V_u = 320$
14. $u = 80$, $v = 20$ and $j = 5$  
    $V_u = 40$, $V_v = 12$, $(V_u V_v) + V_u = 520$

231
<table>
<thead>
<tr>
<th>$\forall u$</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>56</td>
<td>96</td>
<td>176</td>
<td>320</td>
</tr>
<tr>
<td>20</td>
<td>91</td>
<td>156</td>
<td>286</td>
<td>520</td>
</tr>
<tr>
<td>40</td>
<td>161</td>
<td>276</td>
<td>506</td>
<td>920</td>
</tr>
<tr>
<td>80</td>
<td>287</td>
<td>492</td>
<td>902</td>
<td>1,640</td>
</tr>
</tbody>
</table>

(a) Size of Join Site: 5 dyads

<table>
<thead>
<tr>
<th>$\forall u$</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>156</td>
<td>286</td>
<td>520</td>
</tr>
<tr>
<td>40</td>
<td>276</td>
<td>506</td>
<td>920</td>
</tr>
<tr>
<td>80</td>
<td>492</td>
<td>902</td>
<td>1,640</td>
</tr>
</tbody>
</table>

(b) Size of Join Site: 10 dyads

Figure 5.9 Predicted Figures for Maximal Join Operation

15. $u = 80$, $v = 40$ and $j = 5$  
   $V_u = 40$, $V_v = 22$, $(V_u V_v) + V_u = 920$

16. $u = 80$, $v = 80$ and $j = 5$  
   $V_u = 40$, $V_v = 40$, $(V_u V_v) + V_u = 1640$

17. $u = 20$, $v = 20$ and $j = 10$  
   $V_u = 12$, $V_v = 12$, $(V_u V_v) + V_u = 156$

18. $u = 20$, $v = 40$ and $j = 10$  
   $V_u = 12$, $V_v = 22$, $(V_u V_v) + V_u = 276$

19. $u = 20$, $v = 80$ and $j = 10$  
   $V_u = 12$, $V_v = 40$, $(V_u V_v) + V_u = 492$

20. $u = 40$, $v = 20$ and $j = 10$  
   $V_u = 22$, $V_v = 12$, $(V_u V_v) + V_u = 286$

21. $u = 40$, $v = 40$ and $j = 10$  
   $V_u = 22$, $V_v = 22$, $(V_u V_v) + V_u = 506$

22. $u = 40$, $v = 80$ and $j = 10$  
   $V_u = 22$, $V_v = 40$, $(V_u V_v) + V_u = 902$

23. $u = 80$, $v = 20$ and $j = 10$  
   $V_u = 40$, $V_v = 12$, $(V_u V_v) + V_u = 520$

24. $u = 80$, $v = 40$ and $j = 10$  
   $V_u = 40$, $V_v = 22$, $(V_u V_v) + V_u = 920$
25. \( u = 80, v = 80 \) and \( j = 10 \)  \( V_u = 40, V_v = 40, (V_u V_v) + V_u = 1640 \)

These figures are represented graphically in Figures 5.9 and 5.10.
<table>
<thead>
<tr>
<th>v \ u</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>20</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td>40</td>
<td>0.01</td>
<td>0.02</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>80</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>0.06</td>
</tr>
</tbody>
</table>

(a) Size of Join Site: 5 dyads

<table>
<thead>
<tr>
<th>v \ u</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.08</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>40</td>
<td>0.06</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>80</td>
<td>0.04</td>
<td>0.05</td>
<td>0.08</td>
</tr>
</tbody>
</table>

(b) Size of Join Site: 10 dyads

Figure 5.11 Test Figures for Maximal Join Operation

5.3.5.3 Results and Analysis

The twenty-five tests described above produced the results shown in Figure 5.11; Figure 5.12 is a graphical representation of those results.

It can be seen that there is a surprisingly good correspondence between the predicted and actual graphs, leading to the conclusion that the mathematical analysis of the algorithm is generally correct.
5.3.6 Copy

The operation of copying a complex conceptual graph \( u \) to another graph \( v \) is a straightforward duplication of all the contexts, concepts, relations, and lines of identity of \( u \) into \( v \). Most of the work done by the copy operation goes in the copying of simple conceptual graphs, in which the size of the simple graph \( g \) is the predominant factor and the value that shall be varied in order to test the efficiency of copying.
5.3.6.1 Description of Testing

As with other tests in this chapter, a progressive doubling of the size of $u$ (in terms of dyads) has been chosen, giving a series of five tests:

1. $u = 5$
2. $u = 10$
3. $u = 20$
4. $u = 40$
5. $u = 80$

Each test uses a test program to generate $u$ from repetitions of five-dyad basic units. Once $u$ has been formed, the time taken for a copy of $u$ to be made is measured.

5.3.6.2 Predicted Results

Correctness is a simple matter - it will be shown if each test successfully duplicates $u$ and is capable of displaying the duplication.

Efficiency depends on a complex set of structural circumstances. Looking at Algorithm 3.2 of Chapter 3, it can be seen that two loops processing any contexts $\chi$ of $u$ are performed - to copy the relevant structures, and to work out which contexts in the copy of $u$ dominate which others. This is then followed by the main body of the algorithm, which - for each context $\chi$ in $u$ and for each simple graph $\gamma$ within each context - initially visits all concepts $V$ of $u$, before performing another visit and a visit to each edge $E$ of each $V$. Finally, each concept in $u$ is visited for a last time, in order to copy lines of identity.

Based on this algorithm, it is predicted that the running time of the copy algorithm will be proportional to the following, rather complicated, formula:
\[
2\chi + \left( \sum_{i=1}^{\chi} (\sum_{j=0}^{\gamma} 3V_j + E_j) \right)
\]

The copying of simple graphs assumes that \( \chi = 1 \) and \( \gamma = 1 \); these are constant and can be ignored in the following tests. For simple graphs, the formula will simply be:

\[
3V_j + E_j
\]

<table>
<thead>
<tr>
<th>( u )</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>31</td>
</tr>
<tr>
<td>20</td>
<td>56</td>
</tr>
<tr>
<td>40</td>
<td>106</td>
</tr>
<tr>
<td>80</td>
<td>200</td>
</tr>
</tbody>
</table>

(a) Predicted Figures

(b) Graphical Interpretation

Figure 5.13 Predicted Results for Graph Copying
<table>
<thead>
<tr>
<th>u</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>0.03</td>
</tr>
<tr>
<td>10</td>
<td>0.03</td>
</tr>
<tr>
<td>20</td>
<td>0.03</td>
</tr>
<tr>
<td>40</td>
<td>0.04</td>
</tr>
<tr>
<td>80</td>
<td>0.04</td>
</tr>
</tbody>
</table>

(a) Test Figures

(b) Graphical Interpretation

Figure 5.14 Test Results for Graph Copying

The following is therefore predicted:

1. \( u = 5 \): \( V = 4, E = 5, 3V+E = 17 \)
2. \( u = 10 \): \( V = 7, E = 10, 3V+E = 31 \)
3. \( u = 20 \): \( V = 12, E = 20, 3V+E = 56 \)
4. \( u = 40 \): \( V = 22, E = 40, 3V+E = 106 \)
5. \[ u = 80: \quad V = 40, \quad E = 80, \quad 3V + E = 200 \]

These figures are illustrated graphically in Figure 5.13.

5.3.6.3 Results and Analysis

The five tests described above produced the results shown in Figure 5.14(a), whilst Figure 5.14(b) is a graphical representation of those results.

It can quickly be seen that the predicted and actual graphs do not match particularly well - the actual results do not appear to increase as quickly as our interpretation of the copy algorithm would suggest, and this seems to be the case over multiple executions of the test programs. Instead of \(3V + E\), the actual results tend to suggest a growth in running time that is closer to \(V\). Why this should be is uncertain; it may be that an overhead associated with the construction of contexts (even a simple graph needs a zero context and associated structures) is responsible for the early figures (such as \(u = 5, \ u = 10\), and becomes less significant as \(u\) increases.

5.3.7 \(\phi\_A\)

As may be recalled, the operation of \(\phi\_A\) over a domain-extended conceptual graph results in a series of lists that are used to form instructions that may be executed by the underlying Oracle database management system.

5.3.7.1 Description of Testing

\(\phi\_A\) takes the mappings of a domain-extended conceptual graph \(u\) as its primary input, and so any test should vary the number of mappings (and, consequently, the number of dyads) within \(u\). A related issue is the organisation of those mappings within \(u\) - the number of subgraphs into which the set of mappings fragment \(u\) almost certainly has a bearing on the running time of \(\phi\_A\), and any test should also vary this factor.
A series of ten tests are used to evaluate $\phi_\Delta$ - these tests vary both the graph size $d$ (in number of dyads) of $u$, and the number of subgraphs $n$ into which $u$ is fragmented in mapping terms.

1. $d = 10$, $n = 10$
2. $d = 20$, $n = 10$
3. $d = 20$, $n = 20$
4. $d = 40$, $n = 10$
5. $d = 40$, $n = 20$
6. $d = 40$, $n = 40$
7. $d = 80$, $n = 10$
8. $d = 80$, $n = 20$
9. $d = 80$, $n = 40$
10. $d = 80$, $n = 80$

5.3.7.2 Predicted Results

Correctness of the algorithm will be shown if the correct lists $V$, $A$, $S$, $J$ and $R$ are returned. To show the lists returned for each test would take up a lot of space and serve no real purpose; instead, the lists returned for the first test, where $d = 10$ and $g = 10$, are shown.

\[ V = \{C1_1, C2_2, C3_3, C4_4, C1_5, C2_6, C3_7\} \]
\[ A = \{(T1, L1), (T2, L2), (T3, L3), (T4, L4), (T5, L5), (T6, L3), (T7, L1), (T8, L2), (T9, L4), (T10, L5)\} \]
\[ R = {} \]

Figure 5.15 A Sample $\phi_\Delta$ Test Prediction
If each dyad represents a separate subgraph (where each relationship maps to the same relation scheme - for instance, the relationship (R5) always maps to the scheme L5), then it is predicted that the lists returned for this test are those in Figure 5.15. This sample prediction will be considered when examining the actual results later on.

As regards the efficiency of \( \phi_\Delta \), it is sensible to assume that, as both the size of \( u \) and its subgraphs increases, the running time of \( \phi_\Delta \) will increase proportionally. Looking at Algorithm 3.6 of Chapter 3, it can be seen that the number of subgraphs \( g \) of \( u \) govern the initial stages, making an alias and list entries for each attribute \( a \) of the current subgraph. The latter stages are governed by the number of concepts \( V \) in \( u \), with lines of identity being sought.

Based on this algorithm, it is predicted that the running time of \( \phi_\Delta \) is proportional to the following formula:

\[
g \sum_{i=1}^g a_i + V
\]

The following is therefore predicted:

1. \( d = 10, n = 10 \) \((a = 20 + V = 7) = 27\)
2. \( d = 20, n = 10 \) \((a = 26 + V = 12) = 38\)
3. \( d = 20, n = 20 \) \((a = 40 + V = 12) = 52\)
4. \( d = 40, n = 10 \) \((a = 40 + V = 22) = 62\)
5. \( d = 40, n = 20 \) \((a = 52 + V = 22) = 74\)
6. \( d = 40, n = 40 \) \((a = 80 + V = 22) = 102\)
7. \( d = 80, n = 10 \) \((a = 58 + V = 40) = 98\)
8. \( d = 80, n = 20 \) \((a = 100 + V = 40) = 104\)
9. \( d = 80, n = 40 \) \((a = 120 + V = 40) = 160\)
10. \( d = 80, n = 80 \) \((a = 160 + V = 40) = 200\)

These figures are illustrated graphically in Figure 5.16.
### Results and Analysis

This operation operates correctly - as an example, Figure 5.17 is the instruction formed where $u = 10$ and $g = 10$, and conforms to the lists shown in Figure 5.15. As regards efficiency, the ten tests described above produced the results shown in Figure 5.18(a), whilst Figure 5.18(b) is a graphical representation of those results. Figure 5.16 suggests that running time increases as $u$ and $g$ increase, with $g$ being the more important variable in this increase.

#### (a) Predicted Figures

<table>
<thead>
<tr>
<th>$g \cdot u$</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>27</td>
<td>38</td>
<td>62</td>
<td>98</td>
</tr>
<tr>
<td>20</td>
<td>xx</td>
<td>52</td>
<td>74</td>
<td>104</td>
</tr>
<tr>
<td>40</td>
<td>xx</td>
<td>xx</td>
<td>102</td>
<td>160</td>
</tr>
<tr>
<td>80</td>
<td>xx</td>
<td>xx</td>
<td>xx</td>
<td>200</td>
</tr>
</tbody>
</table>

(b) Graphical Interpretation

**Figure 5.16 Predicted Results for $\phi_{N}$**

5.3.7.3 Results and Analysis

This operation operates correctly - as an example, Figure 5.17 is the instruction formed where $u = 10$ and $g = 10$, and conforms to the lists shown in Figure 5.15. As regards efficiency, the ten tests described above produced the results shown in Figure 5.18(a), whilst Figure 5.18(b) is a graphical representation of those results. Figure 5.16 suggests that running time increases as $u$ and $g$ increase, with $g$ being the more important variable in this increase.
Select Instruction : SELECT UNIQUE C1_C1_1, C2_C2_2, C3_C3_3, C4_C4_4, C5_C1_5, C6_C2_6, C7_C3_7 FROM V1

Time taken to compute Phi-Delta instructions for Dyad Size 10, SubGraphs 10:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>real:</td>
<td>0.3100000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>user:</td>
<td>0.0400000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys:</td>
<td>0.0300000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>child user:</td>
<td>0.0000000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>child sys:</td>
<td>0.0000000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.17 SQL Instruction of Figure 5.15

It can be seen from Figure 5.18(b) that the increase in $u$ is generally similar to that predicted, but the increase in $g$ is more significant to running time than had been predicted - for example, where $u = 80$ and $g = 80$. The trend is similar enough to consider the mathematical evaluation at least useful in part, however.
5.3.8 Posting

The computation of a new set of relation schemes from an old set can occur quite frequently in C-GRASS. Sometimes, assertion of new information by the user gives rise to new cardinality or membership information that affects the storage structures of the domain, and it is necessary in these circumstances to recompute the application domain mappings and the storage schemes to which they map.
5.3.8.1 Description of Testing

Posting is a fairly complicated operation, and is difficult to measure effectively because of the large number of variables involved (such as the size of the application domain, the number of cardinalities and memberships already known, the number of changes in that information to be accounted for when recomputing storage, and so on). Therefore, no explicit performance figures are given for this operation; instead, two tests are described that both take an old domain and a modified domain, and which:

- generate the instructions that Oracle needs to create new storage space.
- copy over data from existing (and now redundant) schemes into the new storage.

5.3.8.2 First Posting Test

The first posting test operates over a small library domain. In this domain, a published thing (PT) always has:

- a year of publication.
- an ISBN number.
- one or more titles.
- a publishing company, that always has a name.
- one or more authors, who all have names.
- one or more copies of the original published thing; each copy has a place (a library) where it is located, and each place has its own name.

This sample domain is represented by the domain-extended conceptual graph of Figure 5.19. For expositional purposes, assume that no cardinality or membership information exists in this domains, and that in consequence, all relation schemes to which the domain maps are 2-adic, and that all cardinalities are optional N-M optional. Also note that, as the actual concept labels are being used for clarity, the domain names and scheme names used in Chapter 2 have been dispensed with.
(a) Domain-Extended Conceptual Graph

L1 (COPY PLACE)  L2 (PLACE LIBNAME)
L3 (PT COPY)  L4 (PT BOOKTITLE)
L5 (PT COMPANY-ENT)  L6 (PT ISBN)
L7 (PT YEAR-PUB)  L8 (PT PERSON)
L9 (PERSON AUTH-NAME)  L10 (COMPANY-ENT PUBL-NAME)

(b) Relation Schemes For That Graph

Figure 5.19 Test Domain Before Execution of Posting Test #1

246
In Figure 5.19, all relationships but two have mandatory membership - the LOC relationship between COPY and PLACE is optional on PLACE (i.e.: not every place has to have a copy of a published thing), and the ATTR relationship between PUBLISHED-THING and YEAR-PUBLISHED is optional on YEAR-PUBLISHED (i.e.: conceivably, there may be years when no published things are actually published).

CREATE TABLE L38P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL, C3 CHAR(25) NOT NULL, C4 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L38P (C1, C2, C3, C4) SELECT L7P.C1, L7P.C2, L5P.C2, L6P.C2 FROM L7P, L5P, L6P WHERE L7P.C1 = L5P.C1 AND L7P.C1 = L6P.C1

CREATE TABLE L39P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L39P (C1, C2) SELECT L2P.C1, L2P.C2 FROM L2P

CREATE TABLE L40P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL, UNIQUE (C1, C2))
INSERT INTO L40P (C1, C2) SELECT L8P.C1, L8P.C2 FROM L8P

CREATE TABLE L41P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL UNIQUE, C3 CHAR(25) NOT NULL)
INSERT INTO L41P (C1, C2, C3) SELECT L3P.C1, L3P.C2, L1P.C2 FROM L3P, L1P WHERE L3P.C2 = L1P.C1

CREATE TABLE L42P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L42P (C1, C2) SELECT L9P.C1, L9P.C2 FROM L9P

CREATE TABLE L43P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL, UNIQUE (C1, C2))
INSERT INTO L43P (C1, C2) SELECT L4P.C1, L4P.C2 FROM L4P

CREATE TABLE L44P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L44P (C1, C2) SELECT LIOP.C1, LIOP.C2 FROM LIOP

Figure 5.20 SQL Instructions Produced by Posting Test #1

247
Figure 5.21 Test Domain After Execution of Posting Test #1
The application of the first posting test to this library domain causes the following cardinality and membership modifications:

- [COPY]→(LOC)→[PLACE] becomes mandatory N-1 optional.
- [PLACE]→(NAME)→[LIBNAME] becomes mandatory 1-1 mandatory.
- [PT]→(ATTR)→[COPY] becomes mandatory 1-N mandatory.
- [PT]→(NAME)→[BOOKTITLE] becomes mandatory N-M mandatory.
- [PT]→(ATTR)→[COMPANY-ENT] becomes mandatory N-1 mandatory.
- [PT]→(ATTR)→[ISBN] becomes mandatory 1-1 mandatory.
- [PT]→(ATTR)→[YEAR-PUB] becomes mandatory N-1 optional.
- [PT]→(ATTR)→[PERSON] becomes mandatory N-M mandatory.
- [PERSON]→(NAME)→[AUTH-NAME] becomes mandatory 1-1 mandatory.
- [COMPANY-ENT]→(NAME)→[PUBL-NAME] becomes mandatory 1-1 mandatory.

These changes produce the SQL instructions shown in Figure 5.20, and convert the graph and schemes of Figure 5.19 to those of Figure 5.21.

Posting Test #1 successfully converts the domain represented by Figure 5.19, into a domain that is now normalised with respect to the cardinality and membership changes made to it.

5.3.8.3 Second Posting Test

The second posting test operates over the same small library domain as the first test; the difference is that this test starts out with a slightly modified version of the domain and schemes produced by the first posting test (the table names have been changed slightly). Figure 5.22 represents the domain before the application of the second posting test.

The application of the second posting test to this domain causes the following cardinality and membership modifications:
(a) Domain-Extended Conceptual Graph

L1 (PT COPY PLACE)
L2 (PLACE LIBNAME)
L3 (PT BOOKTITLE)
L4 (PT YEAR-PUB COMPANY-ENT ISBN)
L5 (PT PERSON)
L6 (PERSON AUTH-NAME)
L7 (COMPANY-ENT PUBL-NAME)

(b) Relation Schemes Mapped To

Figure 5.22 Test Domain Before Execution of Posting Test #2

250
• [COPY]→[LOC]→[PLACE] becomes optional N-1 optional.

These changes produce the SQL instructions shown in Figure 5.23, and convert the graph and schemes of Figure 5.22 to those of Figure 5.24.

CREATE TABLE L48P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL)
INSERT INTO L48P (C1, C2) SELECT L1P.C2, L1P.C3 FROM L1P

CREATE TABLE L49P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L49P (C1, C2) SELECT L2P.C1, L2P.C2 FROM L2P

CREATE TABLE L50P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L50P (C1, C2) SELECT L1P.C1, L1P.C2 FROM L1P

CREATE TABLE L51P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL, UNIQUE (C1, C2))
INSERT INTO L51P (C1, C2) SELECT L3P.C1, L3P.C2 FROM L3P

CREATE TABLE L52P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL, C3 CHAR(25) NOT NULL UNIQUE, C4 CHAR(25) NOT NULL)
INSERT INTO L52P (C1, C2, C3, C4) SELECT L4P.C1, L4P.C3, L4P.C4, L4P.C2 FROM L4P

CREATE TABLE L55P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL, UNIQUE (C1, C2))
INSERT INTO L55P (C1, C2) SELECT L5P.C1, L5P.C2 FROM L5P

CREATE TABLE L56P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L56P (C1, C2) SELECT L6P.C1, L6P.C2 FROM L6P

CREATE TABLE L57P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
INSERT INTO L57P (C1, C2) SELECT L7P.C1, L7P.C2 FROM L7P

Figure 5.23 SQL Instructions Produced by Posting Test #2

251
(a) Domain-Extended Conceptual Graph

L48 (COPY PLACE)
L49 (PLACE LIBNAME)
L50 (PT PLACE)
L51 (PT BOOKTITLE)
L52 (PT COMPANY-ENT ISBN YEAR-PUB)
L55 (PT PERSON)
L56 (PERSON AUTH-NAME)
L57 (COMPANY-ENT PUBL-NAME)

(b) Relation Schemes Mapped To

Figure 5.24 Test Domain After Execution of Posting Test #2
Again, Posting Test #2 successfully converts the domain represented by Figure 5.22 into a domain that is now normalised with respect to the cardinality and membership changes made to it. However, a single membership change causes the full restructuring algorithm to be triggered; this usefully illustrates the problem mentioned in the previous chapter with reference to universal and selective restructuring of storage.

5.4 Correctness and Efficiency of Overall System

It has previously been shown that individual operations of C-GRASS work correctly and reasonably efficiently, and so it only remains to show that these operations contribute to the correctness and efficiency of C-GRASS as a functioning processor. In order to demonstrate the correctness of C-GRASS, Appendix G contains details of the construction of a sample application domain. In this section, this domain is discussed, and it is shown that its construction is indeed correct. Each act of construction within Appendix G is numbered, and individual cases will be referred to in order to highlight a particular point of interest.

For ease of discussion, the operation of C-GRASS has been broken down into four basic areas of assertion, which are discussed before moving on to matters such as Inquiry.

5.4.1 Lattice Construction

In any domain, the ontology is of prime importance. Therefore, it is a rule of thumb in C-GRASS that lattices are the first structures constructed in a domain; before such assertions, the domain is empty. Instructions #1 to #59 illustrate the construction of both type and relation lattices for a domain concerned with familial relationships, and Instructions #1a to #59a show how the lattices slowly build up from the most primitive beginnings to its final form.

A basic fact underlying the construction of lattices in C-GRASS is that they are generally built from the root downwards; thus, Instructions #1 to #9 assert type primitives (that are direct subtypes of the three metatypes in C-GRASS), whilst Instructions #10 to #13 assert type...
nonprimitives. Similarly, Instructions #14 to #25 assert relation primitives (that are direct subrelations of the universal relation), whilst Instructions #26 to #59 assert relation nonprimitives.

An assertion of lattice information always contains a supernode, a subnode, and the sort of arc that links them (as well as the lattice into which it is being asserted, of course). If a supernode is unknown (the universal relation and metatypes are always assumed to be known and primitive), then it is automatically assumed to be primitive; and if a node (either supernode or subnode) is nonprimitive and undefined, then the user is asked whether a definition should be specified (for example, Instructions #10 and #25 show the prompt for a definition in the type and relation lattices respectively).

As has been observed, certain lattices are reserved; the universal nodes and metatypes are previously-cited examples, and the three functional relationships - ARG1, ARG2 and RSLT are further examples. All reserved nodes are protected from user abuse - for instance, Instructions #22 to #24 show failed attempts to assert the functional relationships.

5.4.2 Canonical Model Construction

The assertion of the canonical model in C-GRASS starts, like the assertion of lattice information, with the assertion of a model for primitive types and relations, followed by assertion of more specialised information as and when the rest of the domain allows. Instructions #60 to #101 contain canonical model assertions interspersed with definitions, for reasons which shall be elaborated upon in the following section 5.4.3.

Although it has markers, the canonical model does not show them when displayed (Instruction #60a, for instance). If an assertion disagrees with an earlier model assertion, it is rejected (see the discussion of subtyping within canonical model assertion in Chapter 3 for a rationale).

Canonical model assertions cause storage to be generated - or, at the very least, to test if any new storage needs to be generated. This necessary interaction with secondary storage is the
primary reason why canonical model assertion is so woefully slow in comparison with other macro operations (as can be observed by looking at the test figures associated with the canonical instructions in Appendix G). Some application graph assertions also cause similar problems, for the same reason (see section 5.4.4).

5.4.3 Definition Construction

Assertion of definitions is reasonably simple in C-GRASS. As with the lattice and the canonical model, the simplest definitions are defined first (such as Instruction #67, which defines the basic BLOOD_RELATIVE relation used by many other relation definitions). As more complex definitions are asserted, it is necessary to intersperse the block of definition with nonprimitive canonical assertions (such as Instruction #68, which asserts the canonical behaviour of the just-asserted BLOOD_RELATIVE, and which allows the use of that relation in other definitions like Instruction #69, which defines the CHILD relation).

Instructions #63 to #66 assert the small number of type definitions used in the familial domain, whilst Instructions #67 to #103 assert the larger number of relation definitions and the associated canonical model assertions needed to perform the more specialised definitions.

Although definitions may be asserted at any time after this initial burst, such later assertions must follow the same rules as the earlier assertions - they must be canonical, and defined in terms of types and relationships that, if nonprimitive, possess a definition that permits expansion.

5.4.4 Application Graph Construction

Instructions #104 to #124 deal with the assertion of evaluation graphs. Most of the instructions shown here (#104 to #123) assert negative graphs that modify the cardinality and membership of the application domain relation schemes that were created as a result of primitive canonical model assertions - these instructions modify the relation schemes required for domain storage from:
(STATE PERSON) for the EXPR relation.
(ACT PERSON) for the AGNT relation.
(FAMILY FAMILY_NAME) for the first use of the NAME relation.
(FAMILY PERSON) for the MEMB relation.
(ACT PERSON) for the RESULT relation.
(PERSON PERSON) for the RELATIVE relation.
(PERSON PERSON) for the FRIEND relation.
(PERSON PERSON) for the second use of the NAME relation.
(PERSON MALENESS) for the first use of the GENDER relation.
(PERSON FEMALENESS) for the second use of the GENDER relation.
(PERSON PERSON) for the LEGAL_PARTNER relation.

and:

(PERSON MALENESS) for the first use of the GENDER relation, which has become 1-1.

(PERSON FEMALENESS) for the second use of the GENDER relation, which has also become 1-1.

and also:
which combines the first NAME, MEMB and RESULT relations.

Instruction \#124 is an example of a positive graph that carries information that can be stored within the relation schemes constructed by C-GRASS.

5.4.5 Evaluation of Inquiry

In order to give an idea of the efficiency of the assertion and inquiry macros, a small program to generate a simple application graph and a specifiable number of graphs for storage was constructed and executed. This program, along with an edited version of its output, can be seen at the end of Appendix G. It must be stressed that the graphs produced by the program are not particularly canonical - this is because the canonical checking mechanism of C-GRASS was disabled in order to allow their assertion.

The results shown in Appendix G concern three tests, each of which was based upon a randomly-generated five dyad graph. The first test is a sample, asserting and then retrieving five graphs; the second and third tests assert and retrieve one hundred and one thousand graphs respectively. Only the results from the first and second tests (the output of the third test is too voluminous) are shown. It should be evident from the results in Appendix G that C-GRASS generates quite complex relational queries, and this complexity makes Oracle quite unstable in how quickly it processes those queries - fairly short queries may take longer than long queries, depending on Oracle.

5.5 How Variations in Application Domain May Affect Efficiency

Although C-GRASS will operate correctly over all application domains, variations in the size and complexity of the set of application domains may well give rise to considerable variation in its efficiency.
5.5.1 Lattice Complexity

The efficiency of lattice searching in C-GRASS is generally linear. This is fine, but it may also be observed that the structures used to test lattice efficiency were not particularly deep or bushy. Whilst this is true, it does not take into account the fact that the structures generated were uniformly bushy, where every vertex had the same number of subvertices. This led to the structures generated being composed of considerable numbers of vertices and edges, and this is the real measure of complexity.

The tests used on C-GRASS contained potentially large numbers of vertices, and so C-GRASS is capable of efficient handling bushier and deeper lattices that contain comparable numbers of vertices and edges as the examples used in the tests. The lattice searching of C-GRASS is certainly adequate for most application domains, although certain domains (with large ontologies) may require massive lattices of perhaps half a million vertices (but such domains would cause problems for all general approaches in any case).

5.5.2 Granularity of Underlying Relational Storage

C-GRASS operates best over relation schemes which have as large an arity as possible (although within the limitations of Oracle mentioned in Chapter 4). It actively tries to normalise underlying relational storage schemes in order to minimise the number of scheme joins that have to be made when attempting to answer a query. The larger the size (granularity) of the average relation scheme, the more efficient the operation of $\phi_\Delta$ should be.

If, for whatever reason, an application domain favours a small granularity, then the efficiency of $\phi_\Delta$ decreases - as was seen in the testing of this operation, the greater the number of subgraphs in the graph being analysed, the longer the time required to prepare the lists necessary for the preparation of instructions in SQL.
5.5.3 Cardinality of Underlying Relational Storage Schemes

Whilst C-GRASS operates best over relation schemes that have as large an arity as possible, it also functions best over schemes that also have significantly large tuple cardinalities - large and complex application domains with few tuples (which are the traditional staple of expert systems, as seen in the Introduction) fare badly in a system featuring storage and retrieval to/from storage because of overheads required to compute an instruction for Oracle and to execute it - main-memory systems will perform more satisfactorily, because of their lack of such overheads. But as the number of tuples increases, the significance of the overhead required decreases and relational storage becomes more attractive than main-memory storage.

5.5.4 Complexity of Application Graphs

Finally, the complexity of the information kept in secondary storage is related to the issue of tuple cardinality. A domain composed of a small number of large, dissimilar graphs gives rise to large application graphs, each of which map to relation schemes with low tuple cardinality. Such domains, as implied in section 5.5.3, are inefficiently processed by C-GRASS, which prefers large numbers of disjoint graphs that are broadly similar in shape - because such graphs require a smaller set of application graphs, and are more efficient to project into as well as retrieve secondary instantiations from.

5.6 Commentary on Evaluation

The objectives of this chapter were to offer some indication of both the correctness and efficiency of C-GRASS. It has been demonstrated that C-GRASS operates correctly in both its basic and macro functions, and is close to a correct implementation of the extended conceptual graph theory described in Chapter 2.

The issue of functional and overall efficiency is less clear cut. Although efficiency is important, it should be borne in mind that C-GRASS is, first and foremost, a research tool for investigating
the tractability of interfacing conceptual graphs to a relational database; therefore, efficiency of operation was not the issue uppermost on our minds when this implementation was developed. Taking this fact into consideration, the overall level of efficiency is generally quite acceptable - the major bottlenecks of the system occurring at the interface between the conceptual graph and relational database components. As C-GRASS is generally loose-coupled in its implementation, this shouldn't really come as a surprise (the discussion of the merits and demerits of tight- and loose-coupled systems in the Introduction indicated that such an outcome was probable).

The author's experience in implementing C-GRASS has led to two observations:

- the author concurs with [Bocca 1989] and others in their assertion that loose-coupled systems suffer communication problems due to the lack of component integration.
- following from this, it is believed that, whilst the use of a commercial relational database management system for storage and retrieval is certainly tractable, it is not the most efficient arrangement that is possible.

These observations aside, one of the research aims was to be able to extend the semantic capture of such systems - and C-GRASS certainly succeeds in this respect - although certain domain characteristics may lead to inefficiencies.
It is the belief of the author that the research detailed in this thesis fulfils almost all of the original aims that were laid out in the Introduction.

6.1 Summary of Achievements

In the Introduction, six research aims were specified. How well have these aims been met?

6.1.1 Primary Aim: Conceptual Graphs as a Conceptual Schema Definition Language

The primary aim was to show that:

conceptual graphs can act as an implementation-independent conceptual schema definition language for an implementation of the relational model.

It is believed that this aim has been achieved as, with suitable extensions, conceptual graphs can act in such a capacity; they merely need extensions to handle mappings to the logical schema - which need not concern the user if suitable steps are taken - and so are a representation of pure domain knowledge.

6.1.2 Aim #2: Correspondence Between Conceptual Graphs and the Relational Model

The second aim was:
to find out how close a correspondence conceptual graph theory has with the logic of the relational model, and to find out if conceptual graph theory is formally more expressive than relational manipulation languages.

It has been shown that, as conceptual graph theory stands, it does not possess a full correspondence to the logic of the relational model. A simple extension to the semantics, not the syntax, of concepts participating in dyadic conceptual relations was proposed - which is referred to as role domain mappings - in order to represent conceptual graphs as hypergraphs; mechanisms for mapping such extended conceptual graphs into formulas of the tuple relational calculus were then described. However, not all legal expressions of the tuple relational calculus could be expressed as conceptual graphs, and further extensions to conceptual graph syntax and semantics were required.

Because only a subset of conceptual graph theory formally corresponds to the tuple relational calculus, it can be shown that the extended conceptual graphs described are formally more expressive than the relational algebra.

6.1.3 Aim #3: Use of Extended Conceptual Graphs in a Hybrid System

The third aim was:

to find out if extended conceptual graph theory can be used as the theoretical basis for the construction of a hybrid system based on conceptual graphs and a relational database management system.

As mentioned in the Introduction, there have been perhaps no more than one or two serious attempts to design such a system before this research was undertaken. The implementation of C-GRASS shows that the construction of a hybrid system - based on many parts of the extended conceptual graphs that have been developed - is a tractable aim. Although C-GRASS is not particularly efficient in many respects, it more than adequately demonstrates that the theoretical
extensions proposed are sufficient to construct such a system; the theory behind C-GRASS is certainly more sophisticated than the vast majority of other approaches to integrate the relational model and conceptual graphs, and - unlike almost all of those other approaches - it has been successfully implemented.

Although C-GRASS is inefficient in some areas, it should still perform more satisfactorily than most all non-hybrid conceptual graph processors over application domains with large data sets.

6.1.4 Aim #4: Increased Declarativeness of a Hybrid System

The fourth aim was that:

*any system constructed should require no implementation-specific knowledge of an application domain on the part of the users - ideally, the system should be so declarative that the user would not be able to tell if a database system was being used for storage and retrieval of data.*

In order to achieve this aim, the C-GRASS system has two important components - a single knowledge and specification language that is very declarative, and a system of creating and maintaining relational storage structures without user intervention. The first allows the user to express domain information and queries in conceptual graphs - which requires a certain amount of linguistic sophistication, but is much more preferable than other, less declarative manipulation languages such as SQL; the second removes the user's responsibility for the design of storage structures, as well as the requirement to change them if the demands of the application domain change. The user therefore requires no knowledge of the storage structures, or the access paths required to reach them; only a knowledge of the domain is necessary.

6.1.5 Aim #5: Use of Existing Technology in Hybrid System Construction

The fifth aim was that:
the database management component of any hybrid system constructed should be achieved within the framework of an existing commercial database package.

The database component of C-GRASS is implemented in the Oracle relational database management system. As was surmised, this requirement means that C-GRASS is loose-coupled and is a major reason for the poor performance of C-GRASS in certain areas.

6.1.6 Aim #6: Investigation of Legacy Database Reuse in a Hybrid System

The final aim was:

to investigate if existing legacy databases can have their semantics increased by mapping into any conceptual graph - relational database hybrid implementation developed.

This is perhaps the only major aim where a high degree of success cannot be reported. Although it can be shown that the structures of such legacy systems can be mapped to conceptual graphs - and so have the semantics of their data increased - only an informal system for achieving this has been developed where the development of a formal methodology and tool was hoped for.

6.2 Significance of This Research

As to the significance of these findings, the following areas of this research are considered to be both significant and original.

6.2.1 Extended Conceptual Graphs Formally Map to the Relational Model

A formal mapping between conceptual graph theory and the relational model of data storage has been demonstrated. Previous to this, only a partial and imperfect mapping existed. In order to
achieve such a mapping, conceptual graphs have been modified so as to facilitate bidirectional
translation between the two formalisms. A number of existing areas of theory have been
standardised into a single approach, and added a number of new constructions into the theory.
These modifications allow a subset of conceptual graph theory to be relationally complete.

6.2.2 Practical Implementation of Extended Conceptual Graph Theory

A knowledge engineering system has been developed that interfaces a commercial relational
database management system with a conceptual graph processor using many of the theoretical
modifications developed - it uses algorithms that have been developed for the conversion of
conceptual graphs into relational algebraic instructions, and is the first serious hybrid
implementation based on conceptual graphs.

6.2.3 Very Declarative Nature of Implementation

C-GRASS is virtually completely declarative, to the point where the user need not be aware that
a database system is being used for storage purposes. This is due in large part to the automated
design and maintenance of storage structures performed by C-GRASS.

6.3 Unresolved Issues and Future Directions

Although C-GRASS is based on a formal mapping between conceptual graph theory and the
relational model, existing database implementations have gone beyond the basic relational
model and possess many features not found therein. Extension of the research described in this
thesis to encompass such features will give rise to additional areas of research.

In addition, there are a number of areas in the implementation of C-GRASS and its underlying
theory that have not yet been developed fully. This section aims to highlight these undeveloped
areas, and also to point the way towards probable extensions to C-GRASS.
6.3.1 Some Unresolved Issues in Extended Conceptual Graph Theory and C-GRASS

Here are presented what seem to be the most important areas of existing work that need further development.

6.3.1.1 Reasoning Mechanisms

Hybrid systems have a long pedigree (an early sort of hybrid system can be found in [Minker 1978], for example), and many hybrid systems than are more efficient than C-GRASS have been developed. However, C-GRASS differs from these systems in that its use of conceptual graph theory as an artificial intelligence component permits the use of Peirce logic as a reasoning mechanism. Although hybrid languages such as Prolog and Datalog are logically equivalent, conceptual graphs and Peirce logic can express logical rules and constraints far more simply and efficiently than those languages (as seen in Chapter 2). Although C-GRASS does not possess a working Peirce logic processor, it does exist - however, it was not considered sufficiently robust for inclusion in the current system. Further developments should rectify this omission and substantially increase the deductive power of the system.

6.3.1.2 Limitations of Constraints

Although C-GRASS can model static constraints on data, the theory has not been developed to the point where dynamic constraints on data could be modelled - however, they don't formally exist in standard conceptual graph theory either. This would probably require substantial redevelopment of the modified conceptual graph theory described in Chapter 2.

6.3.1.3 Algorithmic Comparison with Hypergraphs

The algorithms for creating relational algebraic calls were developed independently of the algorithms based on the join trees of hypergraphs - it would be interesting to see if the algorithms of C-GRASS are similar to the algorithms used by hypergraphs. If the algorithms
developed for hypergraphs are superior, their adaptation to extended conceptual graph theory would be useful in increasing the efficiency of the system.

6.3.2 Future Directions

Here are presented what seem to be the areas of future research that would be most beneficial to the further development of C-GRASS and its theoretical basis.

6.3.2.1 Additional Database Capabilities

Although C-GRASS possesses query and assertion mechanisms, it would be useful if future versions of C-GRASS also possessed the other two database operational primitives - a method of deleting information, and a method of updating information, and neither of these matters is addressed in conceptual graph theory or in large areas of logic. Although [Fagin et al 1986] have done some work on the matter of updating, deletion is perhaps the more tractable of the two omissions. Some additional work on the logical basis of aggregate functions would also be valuable, in order that they might have a principled implementation in future versions.

6.3.2.2 Legacy Database Systems

Although the development of a formal methodology for importing existing legacy data into C-GRASS was unsuccessful, it would still be useful to find out if the informal system for dealing with mapping legacy database data into C-GRASS can be formalised and completed.

6.3.2.3 Parallelism

As mentioned in Chapter 4, some preliminary research has been done into using a special-purpose parallel transputer array in order to speed up key operations such as projection\(^1\). The additional semantic requirements of extended conceptual graph theory, combined with chronic

\(^1\) We include some notes on this topic in Appendix C.
communication difficulties between the two systems, have meant that such research has been
cursory at best. However, the possibility of integrating a powerful parallel implementation into a
hybrid system is very appealing, and further work in the area would be significant.

6.3.2.4 Improvements to Restructuring Algorithms

As was noted in Chapter 4, the schema restructuring algorithm employed in C-GRASS is less
satisfactory than it might be. A future version of C-GRASS would benefit from a more precise
and specific method of schema restructuring.

6.3.2.5 Adoption of Lower-Level Communications Between Hybrid Components

At present, C-GRASS constructs instructions in SQL that would be, if they were visible,
intelligible to the user. Such instructions have to be parsed and evaluated by the database
management system, and this is an overhead that could be removed from future versions of C-
GRASS if available lower-level communication pathways between the hybrid components were
to be employed.

6.3.2.6 Linkage of Research to ISO Reference Model

The ISO Reference Model (ISO 10746) is a high-level standard concerned with the description
of database domains. Objects within the model possess properties such as abstraction, functional
encapsulation, modularity and so on. What an object is actually depends to a large extent on the
application domain - an object might be a “real-world thing” [de Meer 1995], the denotation of
a model, etc. The Reference Model can map to specification techniques such as SDM and Z (for
example, [Najm and Stefani 1995] demonstrates a typed language with subtyping, whilst [van
Sinderen et al 1995] uses the Reference Model as a basis for a Petri net-like language for
modelling actions and causal relationships), and so it might be interesting to investigate how the
model maps to conceptual graphs in general and this research in particular.
6.4 Coda

C-GRASS was developed as a research tool to test the feasibility of interfacing conceptual graphs and a relational database management system, so it is not a robust product.

As the test results show, it could also be more efficient - particularly in the area of adaptation of storage. Further development and testing of the system is required if it is to achieve its full capabilities.

This thesis represents an important - and practical - step towards the integration of conceptual graphs and relational databases. However, there are still many areas where theory is incomplete, and that gaps still exist between existing theory and implementation. These areas should be the targets of future research.


Date, C. J. (1982), An Introduction to Database Systems (2nd Edition), Addison-Wesley.

Date, C. J. (1990), An Introduction to Database Systems (5th Edition), Addison-Wesley.


273


275


When cardinality or membership constraints are altered in the application domain, it may be that the relation schemes that map to conceptual graphs may also need to be altered, in order to preserve some level of efficient normalisation. We refer to the circumstances under which some change takes place as posting events. There are three classes of event - 'Transitive' Join Dyads (where the two dyads involved are transitively linked), 'Sink' Join Dyads (where the two dyads involved both point into a common concept), and 'Source' Join Dyads (where the two dyads involved both point out of a common concept).

The first step in testing for such events is to check the cardinalities of the various dyads being tested. For each class and event within that class, the following graphs indicate whether the event is a potential success or a simple failure. If the event is potentially successful by virtue of the cardinalities involved, the next step is to check memberships; in one-to-one cardinalities, all memberships must be mandatory for the event to be successful, whilst in one-to-many or many-to-one cardinalities, the membership of the N-cardinality concept must be mandatory. If a schema-joining event is successful, the two relation schemes which mapped to the dyads concerned are merged together into a single scheme.

In the following diagrams, arrows represent directed arcs of conceptual graphs; as labels of relations are not important to this discussion, they have been omitted in the interests of clarity. The arcs have also been modified - for the purposes of exposition only - to show cardinality information:

- represents a one-to-one cardinality;
- represents a one-to-many cardinality;
- represents a many-to-one cardinality;
First Class - ‘Transitive’ Join Dyads

(1) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_2 \ T_3) \rightarrow (T_1 \ T_2 \ T_3)\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\rightarrow & & \rightarrow \\
& & \\
T_3 & & \\
\end{array}
\]

(2) Unsuccessful Schema Join: \((T_1 \ T_2) \circ (T_2 \ T_3) \rightarrow \times\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\rightarrow & & \rightarrow \\
& & \\
T_3 & & \\
\end{array}
\]

(3) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_2 \ T_3) \rightarrow (T_1 \ T_2 \ T_3)\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\rightarrow & & \rightarrow \\
& & \\
T_3 & & \\
\end{array}
\]

(4) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_2 \ T_3) \rightarrow (T_1 \ T_2 \ T_3)\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\rightarrow & & \rightarrow \\
& & \\
T_3 & & \\
\end{array}
\]

(5) Unsuccessful Schema Join: \((T_1 \ T_2) \circ (T_2 \ T_3) \rightarrow \times\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\rightarrow & & \rightarrow \\
& & \\
T_3 & & \\
\end{array}
\]
(6) Potentially Successful Schema Join: $(T_1 \; T_2) \circ (T_2 \; T_3) \rightarrow (T_1 \; T_2 \; T_3)$

(7) Unsuccessful Schema Join: $(T_1 \; T_2) \circ (T_2 \; T_3) \rightarrow \times$

(8) Unsuccessful Schema Join: $(T_1 \; T_2) \circ (T_2 \; T_3) \rightarrow \times$

(9) Unsuccessful Schema Join: $(T_1 \; T_2) \circ (T_2 \; T_3) \rightarrow \times$

Second Class - 'Sink' Join Dyads

(10) Potentially Successful Schema Join: $(T_1 \; T_2) \circ (T_3 \; T_2) \rightarrow (T_1 \; T_2 \; T_3)$
(11) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow (T_1 \ T_2 \ T_3)\)

(12) Unsuccessful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow \times\)

(13) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow (T_1 \ T_2 \ T_3)\)

(14) Potentially Successful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow (T_1 \ T_2 \ T_3)\)

(15) Unsuccessful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow \times\)

(16) Unsuccessful Schema Join: \((T_1 \ T_2) \circ (T_3 \ T_2) \rightarrow \times\)
(17) **Unsuccessful Schema Join:** \((T_1 T_2) \circ (T_3 T_2) \rightarrow \times\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\downarrow & & \downarrow \\
\uparrow & & \uparrow \\
T_3
\end{array}
\]

(18) **Unsuccessful Schema Join:** \((T_1 T_2) \circ (T_3 T_2) \rightarrow \times\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\downarrow & & \downarrow \\
\uparrow & & \uparrow \\
T_3
\end{array}
\]

**Third Class - 'Source' Join Dyads**

(19) **Potentially Successful Schema Join:** \((T_2 T_1) \circ (T_2 T_3) \rightarrow (T_1 T_2 T_3)\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\downarrow & & \downarrow \\
\uparrow & & \uparrow \\
T_3
\end{array}
\]

(20) **Unsuccessful Schema Join:** \((T_2 T_1) \circ (T_2 T_3) \rightarrow \times\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\downarrow & & \downarrow \\
\uparrow & & \uparrow \\
T_3
\end{array}
\]

(21) **Potentially Successful Schema Join:** \((T_2 T_1) \circ (T_2 T_3) \rightarrow (T_1 T_2 T_3)\)

\[
\begin{array}{ccc}
T_1 & \rightarrow & T_2 \\
\downarrow & & \downarrow \\
\uparrow & & \uparrow \\
T_3
\end{array}
\]
(22) Potentially Successful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow (T_1 \ T_2 \ T_3)\)

(23) Unsuccessful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow \times\)

(24) Potentially Successful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow (T_1 \ T_2 \ T_3)\)

(25) Unsuccessful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow \times\)

(26) Unsuccessful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow \times\)

(27) Unsuccessful Schema Join: \((T_2 \ T_1) \circ (T_2 \ T_3) \rightarrow \times\)
It is noticeable that only those events where the join domains are both unique are potentially successful.

The above notation bears a certain resemblance to that found in [Creasy 1992]. However, our use is purely expositional - cardinality and membership information would not be explicitly represented in such structures as shown above.
Appendix B
C-GRASS User Manual

This appendix contains the current manual issued for use with C-GRASS (38 pages).
1. Introduction

C-GRASS is the Conceptual Graph Relational Adaptive Storage System. Unlike other conceptual graph processors developed at Loughborough and elsewhere, C-GRASS is primarily concerned with improving the storage and retrieval of information from a relational database management system, and so much of its functionality is geared to three main tasks:

(a) performing design and maintenance of underlying relational storage in a situation where the application domain is dynamic. Although this aspect of C-GRASS is hidden from the user, the efficient maintenance of the structures used to store the graph-modelled application domain is of prime importance if efficient operation is to be achieved;

(b) using conceptual graphs as a high-level language for database storage and retrieval. This means that the user needs no knowledge of the underlying database manipulation language (there being a translation from conceptual graphs into this language), and also removes the requirement that the user has some knowledge of the database implementation of the
application domain (for example, the user needs to know both the table name and domain name of an item of data in order to retrieve it, and also whether joins are needed between tables in order to get all the data required - this has nothing to do with the application domain per se, just its representation in database terms);

concentrating on letting the user specify what to do, rather than how to do it. Unlike many other graph processing systems, C-GRASS is designed to be as fully automated as possible and, to this end, classical conceptual graph operations such as projection and maximal join are hidden away inside higher level processor commands such as ASSERT and INQUIRY.

C-GRASS is an interface (written in C) between the user and the database; instead of using a relational query language such as QUEL or SQL, C-GRASS instead uses an extended form of the Conceptual Graph knowledge representation format, and the purpose of this document is to outline the form and use of those extensions. In section 2, we look at the additions and changes to conceptual graph syntax and semantics that occur in C-GRASS, whilst sections 3 and 4 deal with the initiation of a C-GRASS session and the command set available to the user within such a session. Section 5 gives an example of a brief session in C-GRASS, and section 6 looks at some of the outstanding problems of C-GRASS. Finally, section 7 contains a list of system messages and a brief explanation of each.

2. Background and Theory

Although the usage of conceptual graphs in C-GRASS is substantially the same as that described in Sowa's text *Conceptual Structures*, there are many differences - some fairly obvious, others rather more subtle. Most of these differences are concerned with matters related to the fact that a secondary storage mechanism is mapped to the conceptual graphs, and modify or extend conceptual graph syntax to deal with this fact; the remaining differences are concerned with eliminating debatable semantic points within the theory.

2.1. Objects in C-GRASS

C-GRASS supports seven sorts of object that can be manipulated by the user:

(a) **Type Hierarchy:** this is mostly the classical Aristotelian type hierarchy, although it contains two important modifications - the ability to support non-Aristotelian arcs without definition, and the support of "metatypes".

The reasoning behind the first modification is grounded in an attempt to establish a position on the tricky matter of multiple supertypes. For example, consider a scenario where the types WATER_VEHICLE, LAND_VEHICLE and AMPHIBIOUS_VEHICLE are subtypes of VEHICLE. This in itself causes no problem, although it would be sensible to make AMPHIBIOUS_VEHICLE a subtype of both WATER_VEHICLE and LAND_VEHICLE; the problem that arises from this situation is one of priority in
definition - which type (LAND_VEHICLE or WATER_VEHICLE) should be used as a genus type for the definition of AMPHIBIOUS_VEHICLE?

The C-GRASS solution is to introduce non-Aristotelian, "logical" arcs between both LAND_VEHICLE and AMPHIBIOUS_VEHICLE, and between WATER_VEHICLE and AMPHIBIOUS_VEHICLE. This allows VEHICLE to provide the genus type (as it is the only type with AMPHIBIOUS_VEHICLE as a direct Aristotelian subtype), whilst still allowing AMPHIBIOUS_VEHICLE to inherit through both LAND_VEHICLE and WATER_VEHICLE.

As for the second modification, any type in the type hierarchy must now be a subtype of one of three metatypes - STRING, NUMERIC, and INDIVIDUAL; a type cannot directly be a subtype of UNIVERSAL in C-GRASS. The reason for this is to allow a principled treatment of numbers and strings as referents, as well as the more usual individual markers. Examples of these metatypes would look like:

- STRING : [FULL_NAME: "John Doe"]
- INDIVIDUAL : [PERSON: #8008]
- NUMERIC : [SALARY_AMOUNT_IN_POUNDS: 16500]

Metatypes are an extension of individual markers to account for numbers and strings (dates were also considered, as well as floats - they may be introduced at a later date). As relations have no "hard" markers, they do not require meta-information, and so (UNIVERSAL) is the top "effective" type in the relational lattice - as opposed to the metatypes in the type lattice.

(b) **Relation Hierarchy:** this is the usual relation hierarchy of conceptual graph theory.

(c) **Type Definitions:** in C-GRASS, these are much the same as in regular theory - except that we limit the body of each definition to a single simple graph.

(d) **Relation Definitions:** as with type definitions, we limit the body of each definition to a single simple graph.

(e) **Function Definitions:** functions are the mechanism by which computation is performed in C-GRASS, and are discussed below. Apart from the restrictions mentioned in that section, functional definition is identical to type definition.

(f) **Second-Order Graph:** the second-order graph (or canonical catalogue, or canonical model) is the principle repository of selectional constraint information in C-GRASS. A entry into the first-order graph may not be performed if it doesn't conform to the canonical model. Any first-order graph that can be formed from application of the canonical formation rules over the canonical model is a canonical graph; the canonical model corresponds to Sowa's set of graphs introduced by insight, and from which others can be formed. However, the canonical model differs from Sowa's concept of canonical graphs in that:
(1) the canonical model consists of a true set and a false set. Within the true set, canonical graphs are joined maximally, and no depth greater than one is permitted within the false set. Therefore, the canonical model consists of what might be regarded as upper and lower bounds on canonicity - those graphs which are specialisations of the true set and generalisations of the false set are canonical, whilst anything else is uncanonical;

(2) canonical formation rules are not explicitly used in checking canonicity of incoming graphs; instead, the incoming graph is broken up into dyadic conceptual graphs (of form \([C1]\rightarrow(R)\rightarrow[C2])\), which are then checked against the canonical model;

(3) unless an entry in the canonical model exists for a relation in an assertion, the assertion cannot proceed; this is because storage for a given dyad is dependent upon its appearance in the canonical model.

First-Order Graph: the first-order graph contains all the information that the user has asserted as graphs, and which aren't selectional constraints like those held in the canonical model. It is composed of two major parts:

(1) the true set, which consists of all new simple graphs joined together as much as possible on identical concepts, and

(2) the false set, which consists of false graphs - both graphs about simple falsity (for example, "P is FALSE") and the complex graphs that represent the laws of the application domain (for example, "if P then Q"). These graphs are true statements about falsity in the domain, so it is more accurate to say that "it is TRUE that P is FALSE", rather than just "P is FALSE".

As C-GRASS is built upon relational storage, it therefore makes sense to strip out data from similar graphs and to store it in relational storage structures. Therefore, the first-order graph is simply a skeleton graph which contains mappings between itself and the storage locations where the instantiations of the graphs are kept. This does not affect assertion or inquiry in any way, although display of the first-order graph will only show this skeleton.

All graphs in both sets are also maximally expanded - this is a consequence of the storage strategy adopted by C-GRASS in order to ease the design and maintenance of relational storage schemes in a system that deals with hierarchical relationships. As with skeletonisation, the only time the user is aware of this strategy is when the first-order graph is displayed.

2.2. Conceptual Graph Syntax and Semantics

The primary modifications to conceptual graph syntax in C-GRASS are as follows:

(a) Primary Syntactic Restrictions: a limit of twenty-five characters has been placed on the length of all labels in C-GRASS - therefore, concepts such as:

\[MY\_GOD\_THIS\_IS\_A\_VERY\_LONG\_LABEL: #300]\n
will be truncated to:
The same restriction goes for referents - nothing over twenty-five characters. More generally, the rule about the final dash in a graph being 'closed' with a period still holds, so the graph:

\[
[SALARY\_IN\_POUNDS: 12500] - \\
(MEAS)\rightarrow[SALARY: #3001] - \\
(ATTR)\rightarrow[PERSON: #502], \\
(MEAS)\rightarrow[SALARY: #3000] - \\
(ATTR)\rightarrow[PERSON: #501].
\]

is acceptable (as the first dash is closed by a period). The graph:

\[
[SALARY\_IN\_POUNDS: 12500] \rightarrow(MEAS)\rightarrow[SALARY: #3001]
\]

is also acceptable, because there is no dash to close and so no need for a period to close it. Because a graph may contain more than one member, the period is used as a simple graph terminator whilst END is used as an overall graph terminator - as in the query:

\[
\text{INQUIRY} \\
[PERSON: *x] - \\
(\text{OFFSPRING})\rightarrow[PERSON] - \\
(\text{OFFSPRING})\rightarrow[PERSON: *y]. \\
\]

This are the primary syntactic restrictions imposed by C-GRASS on conceptual graph syntax; more complicated modifications are detailed in the following subsections.

(b) Lack of λ-Abstractions: although λ-abstractions appear implicitly in the definition of types and relations, C-GRASS does not support their explicit use elsewhere. This is because the point of λ-abstractions is primarily to refer to definitions which do not possess labels, but which the user would like to tread as proper types or relations. For example, the abstraction:

\[
[\lambda x \ [\text{VEHICLE: *x}]\rightarrow(\text{COLR})\rightarrow[\text{REDNESS}] : #300]
\]

could be used in lieu of the graph:

\[
[\text{RED\_VEHICLE: #300}]
\]

assuming the definition:

\[
\text{TYPE RED\_VEHICLE (x) is [VEHICLE: *x]\rightarrow(COLR)\rightarrow[REDNESS] END}
\]
If RED_VEHICLE doesn't appear in the list of definitions, a possible interim solution might be to assign a system-generated label:

[LAMBDA_001: #300]

where:

TYPE LAMBDA_001 (x) is [VEHICLE: *x]→(COLR)→[REDNESS] END

However, this means little to the user if found in the first-order graph. The policy adopted by C-GRASS is based upon the 'limbo' expansion state of such abstractions - they have been expanded, but the expansion hasn't yet been tied in to the rest of the conceptual graph in which it appeared. To illustrate this, consider the graph:

[PERSON: #8)<-(POSS)<-[RED_VEHICLE: #300]

which would - under the maximal expansion regime enforced in C-GRASS - be expanded for storage into:

[PERSON: #8)<-(POSS)<-[VEHICLE: #300]→(COLR)→[REDNESS]

If the graph:

[PERSON: #8] - (POSS)<-[λx [VEHICLE: *x]→(COLR)→[REDNESS]: #300].

is equivalent to:

[PERSON: #8)<-(POSS)<-[LAMBDA_001: #300]

where:

TYPE LAMBDA_001 (x) is [VEHICLE: *x]→(COLR)→[REDNESS] END

it would therefore seem to make sense that maximal expansion would lead this λ-using graph to become:

[PERSON: #8)<-(POSS)<-[VEHICLE: #300]→(COLR)→[REDNESS]

which is the same as the form generated by expansion of the typed concept node. Therefore, we take the view that because C-GRASS enforces a policy of maximal expansion, λ-abstractions are already half-way to this aim and so can be represented in their expanded form without any need for the λ-apparatus:

[λ <parameters> <graph>: <referent>]

then there is no need for such abstractions.
Treatment of Sets: to represent sets, normal conceptual graph theory surrounds the contents of the referent field of a concept with braces, which represents a set of individuals. However, there are a whole raft of drawbacks with such an approach, and these drawbacks seem to stem from the practice of representing set information in the referent field. The most annoying of these is perhaps the inability of the current syntax to specify mixed-type sets. For instance, consider the graph:

\[
\text{[DANCE]} - \\
\text{(AGNT)}\rightarrow\{\text{WOMAN: Jill}\} \\
\text{(AGNT)}\rightarrow\{\text{PERSON: Jack}\}.
\]

Under the rules of set coercion and set join, it would not be possible to form a collective set from this graph, as Jill is a WOMAN but Jack is merely a PERSON. If set join were allowed on this graph, it would end up as:

\[
\text{[PERSON: \{Jack, Jill\}]}\leftarrow\text{(AGNT)\leftarrow[DANCE].}
\]

which has lost information about the womanhood of Jill. Another problem with current set syntax is concerned with the joining of set referents to non-set referents. For example, how can:

\[
\text{[PERSON: \{Jack, Jill\}].}
\]

be joined to:

\[
\text{[PERSON: Jill]} - \\
\text{(AGNT)\leftarrow[EAT]} - \\
\text{(OBJ)\rightarrow[PIE].}
\]

We could coerce Jill to \{Jill\} and set join, but this would give us:

\[
\text{[PERSON: \{Jack, Jill\}]} - \\
\text{(AGNT)\leftarrow[EAT]} - \\
\text{(OBJ)\rightarrow[PIE]},
\]

which is not what was desired at all - instead of meaning that Jill is eating pie, it now means that Jack and Jill are collectively eating pie; the set notation has effectively "bagged" the referent Jill so that relationships that apply to it alone cannot be attached to it. Additionally, current set syntax does not address the nesting of sets in any way. These are quite a simple aspect of set theory, and arbitrarily deep nesting of sets could simply exacerbate the problems mentioned previously. For example, how would a concept like:

\[
\text{[GROUP: \{The Senate, \{Tom, Dick, Harry\}\}]
\]

be processed? Finally, we must consider the effect on the functionality of other graph operations. Regular conceptual graph theory makes no provision for the set syntax or its processing in operations such as projection and join.
Because of all these problems (and many more - the ones outlined are simply the most obvious), the regular set syntax of conceptual graphs has been abandoned. In its place, we favour use of [SET] as a general type - although [SET] itself is not supported explicitly, there is nothing to stop the user adding ANIMATE_SET as a subtype of INDIVIDUAL (and so on). Sets can therefore be defined using abstraction mechanisms:

\[
\text{CAT\_SET} = \lambda x \ [\text{ANIMATE\_SET}: *x] - (\text{MEMB}) \rightarrow \text{[CAT]}.
\]

As sets can appear in the lattice, it is logically acceptable to declare that a CAT\_SET is a subtype of ANIMATE\_SET and so on. Set cardinality can be represented by explicitly declaring all members:

\[
\text{2\_CAT\_SET} = \lambda x \ [\text{ANIMATE\_SET}: *x] - (\text{MEMB}) \rightarrow \text{[CAT]} (\text{MEMB}) \rightarrow \text{[CAT]}.
\]

(d) **Status of Composite Individuals:** at present, there is no explicit support for composite individuals in C-GRASS. This might seem strange in a system based upon database storage, but they are really nothing more than an attractive way of displaying data - one could therefore say that there really is no need for them as they are merely instantiations of definition bodies and can therefore be extracted using regular projection mechanisms.

It would not be difficult to add a command such as DISPLAY INDIVIDUALS MAN END which would display instantiations of type bodies in the way outlined in Sowa; this may appear in a later version of C-GRASS.

(e) **Status of Strings and Numbers:** an effort has been made in C-GRASS to clarify the status of object types such as numbers and strings. Whilst this clarification has been chiefly motivated by the need to know a concept's 'class' in order to properly store it in secondary structures, it was also motivated by a desire to escape from the confusion existing over the difference between WHAT ARE known as LOT\_s and NoLOT\_s - lexical object types and non-lexical object types - that exists in some areas of conceptual graph theory.

The problem is that referents such as John Sowa are often used interchangeably with referents such as #8008. However, the first is really a string that is connected to an individual marker, whilst the second is a proper individual marker. Although it is true that the string can be used as an individual if it uniquely identifies an instance of a concept, C-GRASS avoids this convention and instead encourages the user to adopt the view that strings and numbers are always NoLOT\_s, whilst individuals are always LOT\_s.

To illustrate this, consider the concept:

\[
\text{[SALARY\_IN\_POUNDS: 12500]}
\]

This in itself is not a salary, but a measure of a salary - so if two people both earned £12,500, the graph:
would imply that both people earn the same salary. To be used properly, it ought to be tied
to an individual:

\[
\text{[SALARY\_IN\_POUNDS: 12500] - }
\text{(ATTR)\rightarrow[PERSON: 502]}
\text{(ATTR)\rightarrow[PERSON: 501].}
\]

where two people now earn separate salaries, which just happen to be the same amount.
SALARY is therefore individual and SALARY\_IN\_POUNDS a number. If a NOLOT can
individually identify a concept (such as a unique ID number, for instance), then graphs of
the form:

\[
\text{[PERSON] - }
\text{(ATTR)\rightarrow[ID\_NUMBER].}
\]

are acceptable - such a numeric concept is something unique to the person.

(f) *Monadic and >2-adic Relationships*: C-GRASS supports only dyadic conceptual
relations, so graphs such as the monadic:

\[
\text{(PAST)\rightarrow[TIME]}
\]

and the triadic:

\[
\text{[SPACE] - }
\text{(BETW)1\rightarrow[BRICK]}
\text{(BETW)2\rightarrow[BRICK].}
\]

are not supported - there are two reasons for this lack of support:

1. the number of monadic relations in existence that do not deal with logic
   (such as temporal and modal logical relations) is close to zero, if not
   actually zero;
2. almost all >2-adic relationships can be expressed in binary relational terms.

(g) *No Support for Actors*: in regular theory, actors perform a dual task of performing
computation (in the same way as a dataflow graph) and representing dependencies between
concept node (in the same way as an entity-relationship diagram). In C-GRASS, this dual
purpose is performed by two separate mechanisms - the computational aspect of actors is
now performed by functions (see the following section), whilst the dependency aspect is
now performed by an automatic C-GRASS mechanism that extracts such information from
the rules of the application domain (this means that there is no need for the explicit
representation of dependency).
2.3. Miscellaneous Theoretical Modifications

(a) **Ranges:** when performing inquiries in C-GRASS, any graph containing concepts inheriting from the NUMERIC metatype may be further qualified by the specification of ranges. Such a concept may contain a number of restrictions - for example:

\[ \text{[SALARY\_AMOUNT>5000\&<50000]} \]

will restrict the salaries received to those between £5,000 and £50,000. The range operands available to the user are:

- \( > \) : greater than
- \( < \) : less than
- \( >= \) : greater than or equal to
- \( <= \) : less than or equal to
- \( ! \) : not equal to
- \( = \) : equal to

It should be noted however that if a concept has both a set of range restrictions and a specific referent - as in:

\[ \text{[SALARY\_AMOUNT>5000\&<50000: 27666]} \]

then the referent always takes priority, and so any concept returned will be that with the numeric referent 27666. The syntax of such ranges is fairly rigid, in that they must:

1. always come after the numeric type label of a concept;
2. always place the number after the operand;
3. always be linked together by an ampersand if there are more than one range restriction.

Ranges may not be used outside of graphs in the INQUIRY operation (see the next section).

(b) **Functions:** classical conceptual graph theory uses structures called actors to mediate computation. As actors are effectively types with a computational element (abstraction and definition can be performed for actors), this approach has been rejected in favour of a computation approach using special-purpose types which are circumscribed in the connections that they can have. We refer to these types as functions.

Apart from the element of computation, the main difference between functions and types is that functions have no lattice; although one is theoretically possible, the computational nature of functions makes it undesirable. Like types, functions can be defined in terms of more primitive functions. For example, the function ADD_THREE_NUMBERS might have the definition:
[P_ADD] -
  (ARG1)\rightarrow[NUMERIC]
  (ARG2)\rightarrow[NUMERIC]
  (RSLT)\rightarrow[NUMERIC] -
  (ARG1)\leftarrow[P_ADD] -
  (ARG2)\rightarrow[NUMERIC]
  (RSLT)\rightarrow[NUMERIC],

which uses a number of more primitive functions - the most primitive of which are hardwired into C-GRASS:

P_ADD : Primitive Addition
P_DIVIDE : Primitive Division
P_SUBTRACT : Primitive Subtraction
P_MULTIPLY : Primitive Multiplication

All other functions are ultimately defined in terms of these four primitive functions. As was mentioned above, functions have a very restricted syntax. All functions must look like this:

[FUNCTION] -
  (ARG1)\rightarrow[NUMERIC]
  (ARG2)\rightarrow[NUMERIC]
  (RSLT)\rightarrow[NUMERIC].

All functions are therefore restricted to having two arguments as input and one output, all of which must be a subtype of NUMERIC. If a function has other links than these, or does not possess all of these links, then it is ill-formed.

Functions may be asserted in definitions and first-order graphs, where they occupy the role played by actors. They may also appear in inquiries.

3. Starting a C-GRASS Session

The C-GRASS object code is located in the /longshot subdirectory of /disk07/cobab, on the HPM machine (the password for this account may be obtained from the account owner). To initiate a C-GRASS session, type the command:

cgrass

whilst in the above subdirectory; the following output:
$$ Input, Please

should appear, indicating the establishment of a C-GRASS session. The user may then proceed to assert information into the system, query it, or display information; finally, a session can be ended by typing QUIT (of which more in the following section).

If the user generates an error whilst in C-GRASS, messages of the form:

- $$ Message from Parsing Subsystem...
- $$ No Target (Relation, Type, Function, Catalogue or Graph) specified
- $$ This graph cannot be correctly parsed, and will be rejected

will appear; all system messages are preceded by the $$ sign.

Whenever the system is ready for input from the user, the message:

- $$ What Next?

will be displayed; the system will idle in this state until user input is forthcoming.

4. Operational C-GRASS Command Set

This section is divided into three subsections - the first deals with commands to perform assertions, the second with commands to display information, and the third with other commands. The C-GRASS parser can also support deletions along lines similar to those of the assertion commands, but these are not operational at the time of writing.

4.1. Assertion Commands

(a) Command : ASSERT GRAPH
   <graphs>
   END

Example #1 : ASSERT GRAPH
   [MAN: #333] -
   (OFFSPRING)->[PERSON: #334]
   (OFFSPRING)->[PERSON: #335]
   END
Example #2:

```
ASSERT GRAPH

[MAN: #333] -
  (OFFSPRING)->[PERSON: #334]
  (OFFSPRING)->[PERSON: #335]

END
```

Example #3:

```
ASSERT GRAPH

[PERSON: *x]

[PERSON: *x] -
  (ATTR)->[NUWMBER]

END
```

Result: Examples #1 and #2 will have their data stripped away and placed into secondary storage, with the remaining graph structures being joined to the first-order graph where suitable. Example #3 is appended directly to the false set as a 'rule'.

Comments: expands and adds information to the internal representation of an application domain, with tests to see that those changes are both canonical and consistent. Generally, data can be removed from a graph if either (i) the graph is classically simple, (ii) the graph is simple and contained in an odd context which has no dominated contexts, or (iii) the graph is a combination of (i) and (ii).

Errors: an error will be generated if a graph may contain types or relations that are not yet known to the system; alternatively, it may be uncanonical and therefore will not agree with the selection constraints present in the canonical model. An error will also be generated if the assertion violates a cardinality constraint (for example, every person has a single social security number and each social security number must belong to only one person; the cardinality on the relationship between these two concepts is therefore one-to-one. If the user asserts a graph that violates this constraint - by trying to assert that a person has two such numbers in the same graph, or asserting a number for a person when it is previously known that that person already has a number - then an error is generated). Finally, a graph may introduce inconsistencies into the application domain if added to, and this too will generate an error.

(b) Command:

```
ASSERT CATALOGUE
<graphs>
END
```

Example:

```
ASSERT CATALOGUE
[MAN] -
  (FATHER_OF)<-[PERSON].
END
```

Result: no immediate result visible to user - constrains later assertions.
graphs asserted into the catalogue must be either in the zero context or at depth 1; no nesting or anything more complex than this is allowed.

an assertion into the canonical model may contain undefined types and relations - as this information is currently outside the range of the domain implementation, an error is generated and the graph rejected. An assertion into the canonical model may also duplicate some of the constraints already in the model - if so, an error is generated and it is rejected. Similarly, an assertion may be incompatible with the existing canonical model, and will also be rejected. Finally, an assertion may contain data values as referents - these are only allowed in assertions to the first-order graph and also (in certain circumstances) in assertions of definitions, and so their presence here will cause the graph to be rejected.

(c) Command: `ASSERT TYPE <type name> (<parameter>)
<simple graph>
END`

Example: `ASSERT TYPE TALL_MAN (x)
[MAN: *x] -
    (ATTR)->[TALLNESS].
END`

Result: no immediate result visible to user - need to display definition to see result of assertion.

Comments: this command informs the system of a new type definition.

Errors: if the definition is badly-formed or contains labels not in the type or relation lattices, errors will be generated. Errors will also be generated if the type label already has a definition, if there is an error in the specification of the formal parameter (for instance, the definition may not contain a formal parameter, or contain one that doesn't match the one specified outside the graph). An error may be generated if the genus label doesn't square with the relationship between the two types in the type lattice (for instance, if the arc MAN->TALL_MAN exists in the lattice, the above graph's genus label should be MAN; something like STEAM_ENGINE will generate an error). Finally, an error will be generated if the definition contains non-primitive nodes without their own definitions.

(d) Command: `ASSERT RELATION <relation name> (<parameters>)
<simple graph>
END`

Example: `ASSERT RELATION GRANDFATHER (x, y)
[PERSON: *x] -
    (CHILD)<-[PERSON] -
        (CHILD)<-[MAN: *y].
END`
no immediate result visible to user - need to display definition to see result of assertion.

this command informs the system of a new relation definition. if the definition is badly-formed or contains labels not in the type or relation lattices, errors will be generated. Errors will also be generated if the relation label already has a definition, if there is an error in the specification of the formal parameters (for instance, the definition may not contain any formal parameter, or the wrong number, or contain some that don't match the ones specified in the parameter list). Finally, an error will be generated if the definition contains non-primitive nodes without their own definitions.

(Command: ASSERT FUNCTION <function name> (<parameter>) <simple graph> END

Example: ASSERT FUNCTION ADD_THREE_NUMBERS (x)
[P_ADD: *x] -
(ARG1)->[NUMERIC]
(ARG2)->[NUMERIC]
(RSLT)->[NUMERIC] -
(ARG1)<-[P_ADD] -
(ARG2)->[NUMERIC]
(RSLT)->[NUMERIC]...

END

no immediate result visible to user - need to display definition to see result of assertion.

this command informs the system of a new function definition. Function definitions must be composed solely of other function nodes, numeric nodes, and the three reserved relations ARGument1, ARGument2, and ReSuLT.

any function definition not adhering to the restrictions laid out in the above comments will generate an error; similarly, any function within the definition which is incomplete (for example, lacking a RSLT link) will also generate an error. Finally, a mismatch between the formal parameter in the graph and without (for example, the graph may not contain a genus marker, or the marker may differ between the graph and the parameter list) will generate an error.

(Command: ASSERT LATTICE <supemode> <link symbol> <subnode> END

Example #1: ASSERT LATTICE PERSON >> TALL_PERSON END
Example #2: ASSERT LATTICE VEHICLE > AMPHIBIOUS_VEHICLE END
Example #3: ASSERT LATTICE (UNIVERSAL) >(GENDER) END

no immediate result visible to user - need to look at lattices.

adds a new arc to the type or relation lattice. Link symbol in the above syntax can be either >> (Aristotelian link) or > (Logical link), whilst Example #3 shows that relation labels need to be wrapped in parentheses to distinguish them from the type labels in the other two examples. Given the modifications to the classical
type lattice detailed in section 2, there are various ways in which a lattice link can be asserted into the system. Example #1 adds an "Aristotelian" link between PERSON and TALL_PERSON to the type lattice. If the supertype is not previously known to have existed, the user must specify a metatype (either Individual, String or Numeric) which the supertype will become a primitive of. If either type label is non-primitive and previously undefined, the user is also offered a chance to specify a definition (in the format specified for ASSERT TYPE above - if the graph can't be correctly parsed, the system offers the user a choice between trying again and giving up on the definition - although the arc will still exist if the latter is chosen). Example #2 adds a "logical" link between VEHICLE and AMPHIBIOUS_VEHICLE to the type lattice. Again, if the supertype is not previously known to have existed, then the user must specify a metatype (either Individual, String or Numeric) which the supertype will become a primitive of. If either type label is non-primitive and previously undefined, the user is also offered a chance to specify a definition (in the format specified for ASSERT TYPE). Example #3 adds a "logical" link between UNIVERSAL and GENDER to the relation lattice. If the superrelation is not previously known to have existed, it will be made primitive automatically. If either relation label is non-primitive and previously undefined, the user is also offered a chance to specify a definition (in the format specified for ASSERT RELATION).

an error is generated if the user tries to mix relation and type labels, or if a type arc links a type directly to the UNIVERSAL type instead of a metatype/regular type label. An error will also be generated if a logical link between two type labels exists before an Aristotelian link, or if both logical and Aristotelian links exist (directly or indirectly) between two type labels. Finally - if an Aristotelian link is asserted into the relation lattice, then it will automatically be cast to a logical link instead.

4.2. Display Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>DISPLAY GRAPH END</th>
<th>Example</th>
<th>as above.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>[PERSON: *1] -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *2] -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *4].</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *3] -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *4],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[PERSON: *5] -</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(OFFSPRING)-&gt;[PERSON: *6]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comments : this command displays all of the first-order graph.
Errors : none.

(h) Command : DISPLAY CATALOGUE END
Example : as above.
Result : [PERSON: *50] - (FATHER)->[MAN: *53]
          (MOTHER)->[WOMAN: *51]
          (OFFSPRING)->[PERSON: *49]
          (GENDER)->[MALENESS: *52]
          (GENDER)->[FEMALENESS: *51]
          (ANCESTOR)->[PERSON: *49]
Comments : this command displays the second-order graph, also known as the
canonical catalogue or canonical graph set.
Errors : none.

(i) Command : DISPLAY TYPE <type name> END
Example : DISPLAY TYPE MAN END
Result : TYPE MAN (*11) is
          [PERSON: *11] -
          (GENDER)->[MALENESS].
Comments : this command displays the definition of any type label in the type
lattice that has a type definition attached to it. The figure in
parentheses (*11) is the formal parameter (genus) of the type
definition.
Errors : if the user commands the system to display an unknown type label,
an error will be generated.

(j) Command : DISPLAY RELATION <relation name> END
Example : DISPLAY RELATION GRANDFATHER END
Result : RELATION GRANDFATHER (*30, *32) is
          [PERSON: *31] -
          (OFFSPRING)<-[PERSON: *30]
          (OFFSPRING)->[PERSON: *32].
Comments : this command displays the definition of any relation label in the
relation lattice that has a relation definition attached to it. The
figures in parentheses (*30 and *32) are the formal parameters
(genera) of the relation definition.
Errors : if the user commands the system to display an unknown relation
label, an error will be generated.

(k) Command : DISPLAY FUNCTION <function name> END
Example : DISPLAY FUNCTION ADD END
Result : FUNCTION ADD (*1617) is
          [P_ADD: *1617] -
          (ARG1)->[NUMBER: *1618]
          (ARG2)->[NUMBER: *1619]
          (RSLT)->[NUMBER: *1620].
4.3. Other Commands

(m) Command : INQUIRY <graphs> END
Example #1 : INQUIRY
            [MAN] -
            (OFFSPRING)->[PERSON] -
            (OFFSPRING)<-[WOMAN].
            END

Example #2 : INQUIRY
            [PERSON: *x] -
            (OFFSPRING)->[PERSON] -
            (OFFSPRING)->[PERSON: *y].

            ( [PERSON: *x] -
            (OFFSPRING)->[PERSON: *y].
            )
            END
Example #3 : INQUIRY

([PERSON: *x].

([PERSON: *x] -

(ATTR)->[NI_NUMBER].

))

END

Example #4 : INQUIRY

([MAN] -

(SURNAME)->[SURNAME_STR]

(EARNER_OF)<-[SALARY] -

(MEAS)->[SAL_AMNT>500&<1000].

) END

Result #1 : set of instantiations of Example #1 - for instance:

([MAN: #30] -

(OFFSPRING)->[PERSON: #566] -

(OFFSPRING)<-[WOMAN: #8010].

Result #2 : set of instantiations of Example #2 - for instance:

([PERSON: #8008] -

(OFFSPRING)->[PERSON: #8050] -

(OFFSPRING)->[PERSON: *200].

) returns either TRUE (if the graph can be proved from existing knowledge), FALSE (if the graph can be refuted from existing knowledge) or UNKNOWN (if the existing knowledge cannot prove the graph one way or the other).

Result #3 : set of instantiations of Example #4, where all salaries are between £5,000 and £50,000. For example:

([MAN: #57] -

(SURNAME)->[SURNAME_STR: "Dayspring"]

(EARNER_OF)<-[SALARY: #600] -

(MEAS)->[SAL_AMNT: 25500].

Comments : this command retrieves all examples of the query graph if a simple graph like that of Example #1 is used; alternatively, a complex graph like that of Example #3 is passed to a theorem prover. The difference between these two routes depends on the contexts in the query graph - the basic rule is that if a graph contains nested contexts, it is passed to the theorem prover; this allows graphs such as Example #2 to be processed by the mechanism of projective extent, as well as classical simple graphs.

Errors : if a type or relation label in the query graph is not recognised, or if the query graph is badly formed, then errors will be generated to this effect.
(n) Command : QUIT
Example : as above.
Result : this command will write all information back out to the repository file, close the links to the Oracle database, and terminate the C-GRASS session.
Comments : none.
Errors : none.

5. Using C-GRASS

We now give an example of a brief session in C-GRASS that will hopefully illustrate its functionality and ease of use. In this section, all user input will be in italics, whilst all system output will be in bold text. The following example session is based on the following small application domain. The object of this section is didactic and is not concerned with complexity or efficiency matters, and so the lattices in this example are extremely simple.

(a) Lattices:

UNIVERSAL >> INDIVIDUAL
UNIVERSAL >> NUMERIC
UNIVERSAL >> STRING
INDIVIDUAL >> CITY
INDIVIDUAL >> COUNTRY
INDIVIDUAL >> PERSON
INDIVIDUAL >> CONTINENT
INDIVIDUAL >> POPULATION
INDIVIDUAL >> ELEVATION_IN_FEET
INDIVIDUAL >> LATITUDE_IN_DEGREES
INDIVIDUAL >> LONGITUDE_IN_DEGREES
STRING >> CITYNAME
STRING >> COUNTRYNAME
STRING >> CONTINENTNAME
STRING >> PERSONNAME

(UNIVERSAL) > (NUMERIC_CHRC)
(UNIVERSAL) > (NAME)
(UNIVERSAL) > (RULER)
(UNIVERSAL) > (PART)
(b) Canonical Model:

[CITY] -  
(NAME)→[CITYNAME] 
(NUMERIC_CHRC)→[POPULATION] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[ELEVATION_IN_FEET] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[LATITUDE_IN_DEGREES] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[LONGITUDE_IN_DEGREES] -  
(MEAS)→[NUMERIC].  
(PART)←[COUNTRY] -  
(NAME)→[COUNTRYNAME] 
(RULER)→[PERSON] -  
(NAME)→[PERSONNAME].  
(PART)←[CONTINENT] -  
(NAME)→[CONTINENTNAME].

(c) First-Order Graph:

[CITY] -  
(NAME)→[CITYNAME] 
(NUMERIC_CHRC)→[POPULATION] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[ELEVATION_IN_FEET] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[LATITUDE_IN_DEGREES] -  
(MEAS)→[NUMERIC].  
(NUMERIC_CHRC)→[LONGITUDE_IN_DEGREES] -  
(MEAS)→[NUMERIC].  
(PART)←[COUNTRY] -  
(NAME)→[COUNTRYNAME] 
(RULER)→[PERSON] -  
(NAME)→[PERSONNAME].  
(PART)←[CONTINENT] -  
(NAME)→[CONTINENTNAME].

Note that the first-order graph is very similar to the canonical model; as all of the graphs in the first-order graph (which represents the application domain, more-or-less) are simple, this should not be surprising; as more complex graphs appear (such as information about which countries border each other, for example), the first-order graph will become more complicated.

(d) Definitions: none are present at this stage as all types and relations are primitive.

Starting up the session in the usual way, the system displays its usual prompt:

21
$\text{C-GRASS: Conceptual Graph Relational Adaptive Storage System}$

$\text{Loughborough University Development Version}$

$\text{Input, Please}$

display graph end

[CITY: *10] -
  (NAME)->[CITYNAME: *11]
  (NUMERIC_CHRC)->[POPULATION: *12] -
    (MEAS)->[NUMERIC: *13],
  (NUMERIC_CHRC)->[ELEVATION_IN_FEET: *14] -
    (MEAS)->[NUMERIC: *15],
  (NUMERIC_CHRC)->[LATITUDE_IN_DEGREES: *16] -
    (MEAS)->[NUMERIC: *17],
  (NUMERIC_CHRC)->[LONGITUDE_IN_DEGREES: *18] -
    (MEAS)->[NUMERIC: *19],
(PART)<-[COUNTRY: *20] -
  (NAME)->[COUNTRYNAME: *21]
  (RULER)->[PERSON: *22] -
    (NAME)->[PERSONNAME: *23],
(PART)<-[CONTINENT: *24] -
  (NAME)->[CONTINENTNAME: *25],

$\text{What Next?}$

inquiry
[continentname] <-[name] <-[continent]
end

[CONTINENTNAME: "Africa"] -
  (NAME)<-[CONTINENT: #1].

...<other instantiations>...

[CONTINENTNAME: "Australia"] -
  (NAME)<-[CONTINENT: #5].

$\text{What Next?}$

inquiry
[continentname: "Europe"] -
  (name)<-[continent] -
    (part)->[country] -
      (name)->[countryname],
end
[CONTINENTNAME: "Europe"] -
(NAME)<-[CONTINENT: #2] -
(PART)->[COUNTRY: #6] -
(NAME)->[COUNTRYNAME: "Britain"]

[CONTINENTNAME: "Europe"] -
(NAME)<-[CONTINENT: #2] -
(PART)->[COUNTRY: #7] -
(NAME)->[COUNTRYNAME: "Eire"]

...<other instantiations>...

[CONTINENTNAME: "Europe"] -
(NAME)<-[CONTINENT: #2] -
(PART)->[COUNTRY: #15] -
(NAME)->[COUNTRYNAME: "Germany"]

$$ What Next?

inquiry
[city] -
(name)->[cityname]
(numeric_chrc)->[elevation_in_feet] -
(meas)->[numeric>5000],

end

[CITY: #888] -
(NAME)->[CITYNAME: Mexico City"]
(NUMERIC_CHRC)->[ELEVATION_IN_FEET: #7999] -
(MEAS)->[NUMERIC: 6892].

$$ What Next?

display lattice end

$$ Type Hierarchy Details

UNIVERSAL has Aristotelian subtype INDIVIDUAL
UNIVERSAL has Aristotelian subtype STRING

...<more links>...

STRING has Aristotelian subtype CONTINENTNAME
STRING has Aristotelian subtype PERSONNAME

$$ Press RETURN to look at the relation hierarchy...

<return or any other key>
$\$ Relation Hierarchy Details

UNIVERSAL has logical subrelation NUMERIC_CHRC
UNIVERSAL has logical subrelation NAME
UNIVERSAL has logical subrelation RULER
UNIVERSAL has logical subrelation PART

$\$ What Next?

assert lattice person >> king end

$\$ The type KING has no definition
$\$ Do you wish to define it (Y/N)?

y

$\$ Please assert a definition for this node
$\$ [of form: ASSERT TYPE KING (<parameter>)) <graph> END]...

assert type king (x)
[person: *x] -
    (ruler)<-[country].
end

$\$ What Next?

display type king end

TYPE KING (*35) is
[PERSON: *35] -
    (RULER)->[COUNTRY].

$\$ What Next?

inquiry
[city] -
    (name)->[cityname]
    (part)<-[country] -
        (name)->[countryname: "Freedonia"].
end

$\$ Message from Query Subsystem...
$\$ No Projections of this graph are available

$\$ What Next?
assert graph
[city: #90] -
  (name)->[cityname: #30]
  (part)<-[country: #53] -
    (name)->[countryname: "Freedonia"]
    (ruler)->[person: #100] -
      (name)->[personname: "Rufus T Firefly"]...
end

$$ Message from Parsing Subsystem...
$$ Marker of incorrect conceptual class
$$ This graph cannot be correctly parsed, and will be rejected

$$ What Next?

assert graph
[city: #90] -
  (name)->[cityname: "Freedonia City]
  (part)<-[country: #53] -
    (name)->[countryname: "Freedonia"]
    (ruler)->[person: #100] -
      (name)->[personname: "Rufus T Firefly"]...
end

$$ What Next?

inquiry
[city] -
  (name)->[cityname]
  (part)<-[country] -
    (name)->[countryname: "Freedonia"]...
end

[CITY: #90] -
  (NAME)->[CITYNAME: "Freedonia City"]
  (PART)<-[COUNTRY: #53] -
    (NAME)->[COUNTRYNAME: "Freedonia"]...

$$ What Next?

quit

The session is now terminated and the user returned to the system prompt. The assertions and definitions added to the domain during the session are retained for future use.
6. Current Limitations of C-GRASS

With the exception of a small number of omissions, C-GRASS is currently fully operational, and the only exception of note is that of an operational proof mechanism. Although almost all of a theorem prover has been developed, it is largely untested - although operations such as the removal of double negation and simplification by falsity are known to function properly. Large sections of the proof mechanism have therefore been deliberately disabled until completion of coding and testing is complete. Future work might also hide the storage strategy of C-GRASS in the remaining places where the user can see its consequences - primarily in the display of the expanded and skeletonised first-order graph. Such changes would not affect the overall running of the system and would be primarily cosmetic in nature.

7. System Messages

C-GRASS generates a large number of messages in order to keep the user informed of any errors or other system happenings of importance. They can be split up into four major groups - arising from:

1. assertion of information,
2. parsing of incoming commands,
3. inquiry of the application domain,
4. storage adaptation,

and a minor group of miscellaneous messages generated by other happenings.

7.1. Assertion Subsystem Messages

$$\text{Genus label TALL\_DONKEY is not in the type lattice}$$
$$\text{C-GRASS cannot store this definition}$$

The definition just asserted contains a genus label that is unknown to the system and so the definition will be rejected until it is added to the relevant lattice.

$$\text{Type label TALL\_DONKEY is not in the type lattice}$$
$$\text{C-GRASS cannot store this definition}$$

The new definition label is unknown to the system and so the definition will be rejected until it is added to the relevant lattice.

$$\text{Lattice relation between DONKEY and TALL\_DONKEY is non-existent/indirect}$$
$$\text{C-GRASS cannot store this definition}$$
When asserting a type definition, the label of the new definition (TAIL_DONKEY) must be a direct Aristotelian subtype of its genus label (DONKEY) in the type lattice; if this is not so, the definition has faulty inheritance and is not stored.

$$ Relation label MATR is not in the relation lattice
$$ C-GRASS cannot store this definition

The new definition label is unknown to the system and so the definition will be rejected until it is added to the relevant lattice.

$$ Is the type DONKEY (I)ndividual, (N)umeric, or (S)tring? Please Specify

If the type DONKEY is acting in the role of a supertype - as in:

```
ASSERT LATTICE DONKEY >> TALL_DONKEY
```

- and is previously unknown, it is assumed, for the time being at least, to be a primitive type and so must be allocated to one of the three metatypes.

$$ Inadmissible metatype - (I)ndividual, (N)umeric, or (S)tring?

Related to the above message, this error is generated if the user does not specify either I, N, or S as a metatype option.

$$ A type cannot be a direct subtype of UNIVERSAL
$$ Try INDIVIDUAL, STRING or NUMERIC instead

Any type asserted must be a subtype of a metatype, in order that the system knows what sort of referents to expect in its instantiations; for a type to directly inherit from UNIVERSAL is not permitted.

$$ Aristotelian arc already exists, directly or indirectly
$$ Therefore, the arc will not be stored

The link just asserted by the user duplicates an existing link in the type lattice: either directly - in which case the new link is superfluous, or indirectly - the asserted line of inheritance can be duplicated by several links. For example, if PERSON is a supertype of TALL_PERSON and TALL_PERSON is a supertype of VERY_TALL_PERSON, then making a direct link between PERSON and VERY_TALL_PERSON is not stored as it can be deduced from existing links.

$$ Logical arc already exists, directly or indirectly
$$ Therefore, the arc will not be stored

The link just asserted by the user duplicates an existing link in one of the lattices: either directly - in which case the new link is superfluous, or indirectly - the asserted line of inheritance can be duplicated by several links. For example, if PERSON is a logical supertype of TALL_PERSON and TALL_PERSON is a logical supertype of VERY_TALL_PERSON, then making a direct
logical link between PERSON and VERY_TALL_PERSON is not stored as it can be deduced from existing links.

$$\text{Multiple Aristotelian links to DONKEY exist - illegal construction}$$

$$\text{Therefore, the arc will not be stored}$$

Fairly self-evident - the last assertion will cause a type to inherit along two Aristotelian lines of inheritance - such as AMPHIBIOUS_VEHICLE inheriting from both LAND_VEHICLE and WATER_VEHICLE. This is not allowed as it causes definitional problems.

$$\text{Aristotelian arc not allowed in relation lattice - casting to logical}$$

Warns the user that only logical links are allowed between nodes of the relation lattice, and so recasts assertions like:

$$\text{ASSERT LATTICE (UNIVERSAL) >> (MATR) END}$$

$$\text{to:}$$

$$\text{ASSERT LATTICE (UNIVERSAL) >> (MATR) END}$$

$$\text{Asserting Logical link where direct/indirect Aristotelian link exists}$$

$$\text{Therefore, the arc will not be stored}$$

An Aristotelian link is a stronger line of inheritance than a logical link and is preferable wherever possible, so if an Aristotelian link can be shown to exist between the two nodes which the user is attempting to link logically, the new logical link is superfluous and is not needed.

$$\text{This definition label already exists}$$

$$\text{Therefore, this definition will not be asserted}$$

The user has tried to define a definition where one is already known; this is not permitted.

$$\text{The asserted graph contradicts existing information;}$$

$$\text{Therefore, this graph cannot be asserted}$$

Theorem proof has found that the previous assertion does not agree with all the information that has gone before. As an inconsistent domain of information must be avoided, the assertion is rejected.

$$\text{The asserted graph is already known or can be inferred;}$$

$$\text{Therefore, this graph will not be stored}$$

Theorem proof has found that the previous assertion either duplicates something that was already directly known (in which case re-storage is redundant), or could be deduced through theorem proof. Either way, the assertion is rejected.
The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Theorem proof cannot prove or disprove the assertion just made - therefore, it is genuinely new information and will be stored.

This assertion contains undefined types/relations; therefore, it will not be asserted.

The asserted graph contains type labels and relation labels that have no definitions; as the efficient storage of conceptual graphs depends on those graphs being maximally expandable, we prevent such assertions until the relevant labels have been given proper definitions.

Function doesn't have right number of mandatory arcs (ARG1, ARG2, RSLT); therefore, this definition cannot be asserted.

Any function use in C-GRASS must have three arcs - ARGument1, ARGument2, and ReSuLT; if a function has links other than these, or doesn't have all the arcs that it should have, then the function is badly-formed and so any definition it appears in is also badly-formed.

The type DONKEY has no definition.

Do you wish to define it (Y/N)?

The type label DONKEY has just featured in a lattice link assertion and is not primitive; as it doesn't have a definition as yet, the system asks the user if one should be defined at this juncture.

Please assert a definition for this node [of form: ASSERT TYPE DONKEY (<parameter>) <graph> END]

Related to the above message - if the user does wish to define a type at this juncture, then this message is displayed, telling the user the form that the definition should take.

The relation GRANDFATHER has no definition.

Do you wish to define it (Y/N)?

The relation label GRANDFATHER has just featured in a lattice link assertion and is not primitive; as it doesn't have a definition as yet, the system asks the user if one should be defined at this juncture.

Please assert a definition for this node [of form: ASSERT RELATION GRANDFATHER (<parameters>) <graph> END]

Related to the above message - if the user does wish to define a relation at this juncture, then this message is displayed, telling the user the form that the definition should take.

Definition didn't conform to template outlined or is for the wrong label.

Select either (Q)uit or any other key to try again:
When entering a definition as a result of referring to a definitionless non-primitive type or relation when asserting a lattice link, the user is asked if he/she wishes to assert a definition for that type or relation (see previous two messages for an instance of this). If this is so, a 'template' ("ASSERT RELATION.....", etc) is displayed for the user to follow, and this error is generated if the input under the circumstances mentioned does not conform to the specified template. The user is offered another chance to get it right, if he or she should so wish.

$$ \text{Definition contains undefined non-primitive nodes} $$
$$ \text{C-GRASS cannot store this definition} $$

A definition must be composed of types and relations that are either primitive (direct subtypes of a metatype, or a direct subrelation of the universal relation) or which have a definition. This is to permit maximal expansion of definitions, and is a consequence of the storage strategy of C-GRASS.

$$ \text{Logical link in type lattice asserted before Aristotelian link} $$
$$ \text{Aristotelian links must be asserted before any Logical links} $$

Given the construction:

\[
\text{PERSON} > \text{TALL\_PERSON}
\]

the assertion of the link:

\[
\text{TALL\_PERSON} > \text{VERY\_TALL\_PERSON}
\]

cannot be accepted in C-GRASS, as there is no direct Aristotelian line of inheritance between \text{TALL\_PERSON} and \text{VERY\_TALL\_PERSON}, only a logical link. If we consider the classic example used previously:

\[
\begin{align*}
\text{VEHICLE} & \rightarrow \text{LAND\_VEHICLE} \\
\text{VEHICLE} & \rightarrow \text{WATER\_VEHICLE} \\
\text{VEHICLE} & \rightarrow \text{AMPHIBIOUS\_VEHICLE} \\
\text{LAND\_VEHICLE} & \rightarrow \text{AMPHIBIOUS\_VEHICLE} \\
\text{WATER\_VEHICLE} & \rightarrow \text{AMPHIBIOUS\_VEHICLE}
\end{align*}
\]

it can be seen that there is a direct Aristotelian link from the metatypes to the most specialised type, and that the logical links serve to sort out difficulties with definition in a multiple inheritance scenario. In the first construction, there is no such direct Aristotelian link, and so it must be rejected.

$$ \text{Defn is uncanonical - at least one number in the definition} $$
$$ \text{is the sink for two or more RSLT relations} $$
$$ \text{therefore, this definition cannot be asserted} $$

When specifying the body of a definition, the user has made the error of letting a numeric concept be the 'target' concept for two functions - for example:
If both functions are capable of instantiation, they will both try to place a value in \( *x \); depending on the order of instantiation, one will overwrite the other. Such a state of affairs cannot be permitted, and so the definition is rejected.

$$ The formerly primitive type DONKEY has become non-primitive
$$ Please assert a definition for this node
$$ [of form: ASSERT TYPE DONKEY (<parameter>) <graph> END]...

If a formerly primitive type label (ie: a direct subtype of a metatype) has been converted into something that is non-primitive, then it is imperative that a definition is obtained for it. Unlike the usual situations in which non-primitive types are added to the lattice, the system cannot afford to defer the acquisition of this definition until later; because if it does, it may lead to inefficiencies in the underlying storage - such that C-GRASS may no longer be able to store all information because it doesn't know how to convert DONKEY into its most primitive form.

If the user cannot assert a definition in this place, the actions that made DONKEY non-primitive are rolled back, and DONKEY becomes primitive again.

$$ The formerly primitive relation ATTR has become non-primitive
$$ Please assert a definition for this node
$$ [of form: ASSERT RELATION ATTR (<parameters>) <graph> END]...

The reasoning behind this message is virtually identical to that of the previous message.

$$ Defn is uncanonical - incorrect use of relation, reflexivity violation,
$$ or canon has no entry for one or more relations in the definition.
$$ Therefore, this definition cannot be asserted
$$ Select either (Q)uit or any other key to try again

The user has attempted to assert a definition with a body that disagrees with the canonical model, or (worse, because it interferes with the evolution of storage structures in the secondary store) has no mention in the canonical model. As the canonical model is the final arbiter on how concepts and relationships are allowed to connect to each other, the assertion loses out and is rejected (although the user gets to make more attempts at definition if desired).

$$ Graph is uncanonical - incorrect use of relation, reflexivity violation,
$$ or canon has no entry for one or more relations in the assertion;
$$ therefore, this graph cannot be asserted
Identical circumstances to the previous message generate this error, save for the fact that graph assertion instead of definition assertion is being undertaken.

$$\text{Assertion into the canonical model contains data values;}
$$ therefore, it will not be asserted

Unlike assertions into the first-order graph, data values cannot be asserted into the canonical model as it has nowhere to store them, and is simply a repository for how concepts and relationships connect together in any case.

$$\text{Assertion into the canonical model is incompatible with previous assertions;}
$$ therefore, it will not be asserted

If the relations in the false set of the canonical model do not have attachments that are subtypes of the corresponding relations in the true set after addition of the incoming canonical graph, this error is generated. For example:

\[
\begin{align*}
\text{CAR} & \rightarrow \text{DRIVER_OF} \rightarrow \text{PERSON} \\
\text{CAR} & \rightarrow \text{DRIVER_OF} \rightarrow \text{PERSON_SANS_LICENCE}
\end{align*}
\]

would be okay, as CAR < CAR and PERSON_SANS_LICENCE < PERSON. All that the system does is look at all the relations in the even enclosure and find out if they have a match in the odd enclosure; if they do, then the attachments are checked - if a non subtype is found, we generate the error.

$$\text{Assertion into the canonical model duplicates existing knowledge}
$$ or is in conflict with a previous canonical assertion about a
$$ relation on the same line of inheritance;
$$ therefore, it will not be asserted

This error is generated if the asserted graph contains information that is already duplicated by the canonical model. For example, say that graph (a) below is already in the canonical model, and that first graph (b) and then graph (c) are asserted:

\[
\begin{align*}
(a) & \quad \text{PERSON} \rightarrow \text{GENDER} \rightarrow \text{MALENESS} \\
(b) & \quad \text{PERSON} \rightarrow \text{GENDER} \rightarrow \text{FEMALENESS} \\
(c) & \quad \text{TALL_PERSON} \rightarrow \text{GENDER} \rightarrow \text{MALENESS}
\end{align*}
\]

It would be permissible to allow the assertion of (b) to proceed, because it doesn't duplicate existing stuff, whereas the assertion of (c) would be rejected because it is a specialisation of (a), and so would duplicate something already known about. Note that if the enclosure is odd, this is more or less the other way around - we would let (c) be asserted, given (a), because it forms a lower bound than (a) does.

$$\text{Assertion into the canonical model contains undefined types/relations;}
$$ therefore, it will not be asserted
The canonical graph just asserted contains types and relations that aren't in the type and relation lattices - it refers to knowledge that is currently outside the internal representation of the application domain and is therefore badly-formed.

$$ The asserted graph violates previously declared cardinality constraints; $$ therefore, this graph will not be stored

This error is generated if the user tries to break the cardinality limit on a relation. For example, every person has a single social security number and each social security number must belong to only one person; the cardinality on the relationship between these two concepts is therefore one-to-one. If the user asserts a graph that violates this constraint - by trying to assert that a person has two such numbers in the same graph, or asserting a number for a person when it is previously known that that person already has a number - then the graph will be rejected.

$$ Functions may only contain numbers or other functions as concepts, $$ alternatively, a function used contains an restricted arc (ARG1, ARG2, RSLT) $$ into it instead of away from it - this is an illegal formation; $$ therefore, this definition cannot be asserted

A function in the input graph does not conform to the strictures that C-GRASS places upon them (see section 2.3 for details). The input is therefore rejected.

$$ Function arguments in the graph are invalid, RSLT protocol violated, $$ or the function does not possess mandatory arcs (ARG1, ARG2, RSLT); $$ therefore, this graph cannot be asserted

This error is generated in similar circumstances to the previous message - if the arguments of a given function are non-numeric, if two or more functions share a common numeric concept as their result 'sink', or if a given function has either the wrong sort of, or too few, connections.

7.2. Parsing Subsystem Messages

$$ No action (Assertion, Inquiry, Deletion, Display or Quit) specified $$ This graph cannot be correctly parsed, and will be rejected

The system couldn't tell what the last command was supposed to do as it didn't contain a system action. As it wasn't sure what was required, it erred on the side of caution and rejected the input.

$$ No target (Relation, Type, Function, Catalogue or Graph) specified $$ This graph cannot be correctly parsed, and will be rejected

Although the system knew what the user wanted to do, it couldn't work out what it was to do it to as the user's input was faulty.

$$ Can't form a valid conceptual graph from input $$ This graph cannot be correctly parsed, and will be rejected
The graph in the input was badly-formed and didn't conform to conceptual graphs syntax as laid out in earlier sections. Apart from this, it doesn't know the particulars of why the graph is badly-formed.

$$ END terminator missing from graph$$ This graph cannot be correctly parsed, and will be rejected

Each action must be ended with an END terminator so that the system knows that the user's input is complete. In this case, that terminator is absent.

$$ Label not specified in input$$ This graph cannot be correctly parsed, and will be rejected

The definition under construction is missing a label, so the system doesn't know what to call this definition; it therefore rejects it.

$$ Error in specification of formal parameters$$ This graph cannot be correctly parsed, and will be rejected

The definition just asserted has a formal parameter error - this means that (a) the body of the definition has a different number of formal parameters to those listed in the parameter list, or (b) the names of the parameters in the list and in the body of the definition don't agree with each other.

$$ Badly-formed relation somewhere in input$$ This graph cannot be correctly parsed, and will be rejected

A relation in the graph is syntactically incorrect - conceptual relations such as - >(MAT$: #3)-> or - >()-> would be rejected.

$$ Badly-formed concept somewhere in input$$ This graph cannot be correctly parsed, and will be rejected

A concept in the graph is syntactically incorrect - for example, concepts such as [:#3] or [PERSON; #300] or [PERSON: zz] would be rejected.

$$ Missing relational arc - can't complete dyadic connection$$ This graph cannot be correctly parsed, and will be rejected

The user has neglected to form a relational arc properly - for example, relational arcs like [BRICK] - >(MATR) >[CLAY] would be rejected.

$$ Can't find relation where one expected$$ This graph cannot be correctly parsed, and will be rejected

The syntax of conceptual graphs has led the system to expect a relation where one cannot be found (generally, in the middle of two arcs). The graph is therefore badly-formed and is rejected.
$\$ Can't find concept where one expected
$\$ This graph cannot be correctly parsed, and will be rejected

The syntax of conceptual graphs has led the system to expect a concept node where one cannot be found (generally at the ends of arcs). The graph is therefore badly-formed and is rejected.

$\$ Context can't be formed, or expected concept is missing
$\$ This graph cannot be correctly parsed, and will be rejected

At some point in the graph, the system observed the termination of a simple graph whilst inside a non-zero context. It therefore expected one of two things: (i) possibly another concept, indicating another simple graph to be parsed; (ii) a parenthesis to close that context. What has happened here is that we are still inside a non-zero context and an END terminator has appeared - the context is therefore not closed and the graph is badly-formed.

$\$ END terminator or expected concept missing, or context can't be formed
$\$ This graph cannot be correctly parsed, and will be rejected

This error is very similar to its predecessor. Either (i) a context was opened and has not been not terminated - there is a left parenthesis ( which is not matched by a right parenthesis ) somewhere in the graph; or (ii) a simple graph has just being correctly parsed and the system is looking for either a new concept or left parenthesis (to indicate a new graph), or the END terminator (to indicate the termination of the graph).

$\$ Unknown type use or function use outside definition (not allowed)
$\$ This graph cannot be correctly parsed, and will be rejected

Although functions may appear outside definitions as a result of maximal expansion of those definitions (they can be found in the first-order graph in this way), explicit use of such structures by the user outside such structures is not permitted (after all, what would be the point of an explicit function appearing in a query graph, for example). Alternatively, the user may just have used an unknown type label in the input - it should be fairly clear from the input which of the two errors has occurred.

$\$ Range-restricted type label has occurred outside a query (not allowed)
$\$ This graph cannot be correctly parsed, and will be rejected

Range restrictions are a function of querying, and the system has observed the user typing to use them somewhere else (in the assertion of a definition, for instance). Although such restrictions might be supported in such places in later versions of the system, they are not currently supported and the input is rejected.

$\$ Unknown type/function label used - please define before use
$\$ This graph cannot be correctly parsed, and will be rejected

The user has simply used a type or function label that is not listed in the type lattice or function list - this cannot be allowed to proceed for obvious reasons.
Unknown relation label used - please define before use
This graph cannot be correctly parsed, and will be rejected

The user has simply used a relation label that is not listed in the relation lattice - as with unknown types or functions, this cannot be allowed to proceed.

Marker of incorrect conceptual class
This graph cannot be correctly parsed, and will be rejected

A concept in the graph has a referent that is inconsistent with its inheritance from a given metatype - for example, [PERSON: "Warren Worthington"] would generate this error if PERSON inherited from any metatype other than STRING.

Range-restricted type label - referent fails the test
This graph cannot be correctly parsed, and will be rejected

This error arises if a numeric concept in a query possesses range restrictions but is also the result concept for a function that appeared upon the maximal expansion of the query graph. If the concept is instantiated, and that instantiation does not agree with the range restrictions, then the error is generated and the graph rejected.

Unknown characters in input string - input rejected
This graph cannot be correctly parsed, and will be rejected

Quite a general message; the user has used a token or character - such as % or DOUGHNUT - that is unfamiliar to the system. Familiar tokens include ASSERT, (, END, and so on.

Attempted to store number outside permissible ranges - input rejected
This graph cannot be correctly parsed, and will be rejected

Any computer-based system has limits on what it can store. In C-GRASS, the largest positive number that can be stored is 9999999999, and the largest negative number that can be stored is -9900000000 (numbers beneath this are used for processing of generic numbers in C-GRASS). As a result of either direct assertion or of function activity, the graph contains a numeric concept with a referent that is outside these bounds, and such overflows cannot be stored.

Grammatical violation - usage of incorrect punctuation (comma, stop)
This graph cannot be correctly parsed, and will be rejected

The user has simply used the wrong punctuation when forming the input. See the earlier section on graph syntax for guidance on this matter.

7.3. Query Subsystem Messages

Requested query is uncanonical;
therefore, it will not be performed
Self-evident - if the query graph has violated selection constraints, there will be no projections of it into a set of graphs that are known to be canonical. The system cuts off the query at this early stage to save processing of something that will fail.

$$ \text{Function arguments in the graph are invalid, RSLT protocol violated,} $$
$$ \text{or function does not possess mandatory arcs (ARG1, ARG2, RSLT);} $$
$$ \text{therefore, this graph cannot be asserted} $$

One or more functions present in the query graph either have connections to non-numeric concepts, or have the wrong number of mandatory arcs or unlawful arcs (anything that isn't mandatory is unlawful). Alternatively, a numeric concept in the query graph may be targeted as the result of two functions, which is mathematically unsound and therefore illegal.

$$ \text{No Projections of this graph are available} $$

No instances of the query graph are stored, and so none can be returned.

$$ \text{The query graph is FALSE} $$

If a complex graph has been used as a query graph, it is passed to the theorem prover for evaluation. If it can be proven that the graph violates existing graphs, then the query is FALSE.

$$ \text{The query is TRUE} $$

If a complex graph has been used as a query graph, it is passed to the theorem prover for evaluation. If it can be proven that the graph agrees with existing graphs, then the query is TRUE.

$$ \text{The query is UNKNOWN} $$

If a complex graph has been used as a query graph, it is passed to the theorem prover for evaluation. If the query graph can't be proved or disproved either way, then the query must be UNKNOWN.

$$ \text{Function instantiations have violated range restriction;} $$
$$ \text{Inquiry has been automatically aborted} $$

The query graph contained functions that were evaluated by the system; as a result, one or more range-restricted numeric concepts have gained values that conflict with those range queries, and so the query has been rejected.

7.4. Adaptive Component Subsystem Messages

$$ \text{Cardinality violation prevented} $$
$$ \text{Attempted modification of the canonical model prevented because it} $$
$$ \text{violated cardinality of existing data. Canonical model rolled back} $$
This message is generated in response to the user's attempt to modify the canonical model with information that violates the 'extensional' cardinality of existing data. For example, suppose that there was no restriction on the number of cars a person could own, and that a number of graphs were asserted to this effect; if a graph is then asserted which limits the number of cars a person can own to just one, then the graphs previously stored violate that new canonical constraint. As we can't throw those graphs away, it is therefore preferable to reject the attempted modification until those graphs have been dealt with.

$$ Canonical deletion prevented
$$ Attempted deletion from the canonical model prevented because it $$ deleted known instantiations. Canonical model rolled back

If the user attempts to remove information from the canonical model, it may give rise to a situation in which certain graphs - which were assigned storage based on that canonical information - must be deleted. The system will not delete information in such an indirect way, and so prevents the removal of the canonical information until those graphs have been dealt with first.

7.5. Miscellaneous Messages

$$ Repository data files appear to be absent
$$ Please run the SETUP

A rare message that may occur on first use of C-GRASS, it tells the user that a number of miscellaneous storage structures required by C-GRASS are not yet present. To set up these structures, run SETUP.

$$ Unknown definition - cannot display

The user has attempted to display the definition of a type which either does not exist in the lattices, or exists but has no definition defined.

$$ The type requested is a metatype - such types have no definitions

The user has attempted to display a definition for a system type - this is not permitted as they are merely types from which certain referential properties are inherited by their subtypes.
Appendix C
Notes on Parallel Extensions to C-GRASS

This appendix contains several small documents generated by the author and a fellow research student, Alfred Chan, who is currently working on a parallel implementation of a number of conceptual graph operations.

1. Design of an Intelligent Parallel Database System: contains some initial ideas on how the two research streams might be combined into a single system.

2. Notes Regarding Data Structure: as the two systems had evolved in isolation of each other, this document attempted to establish some structural translations between the two.

3. Parallel Approach to Efficient Relational Decomposition: this document dates from the period when the methods used to perform scheme restructuring were based on algorithms found in [Ullman 1988], and proposed a parallel approach to the computation of restructuring.

4. PC->Transputer Translator routines: a simple program that was intended to convert C-GRASS structures into parallel structures.

5. Bowen_Chan_Conversion routines: a more sophisticated attempt to perform (4).
Design of an Intelligence Parallel Database System

Physical Diagram of the proposed Database System

A salient feature of this system is that the primary memory of the PC and the primary memory of the Transputers Network (T-N) are not connected directly. To cut down the amount of data traffic the T-N keeps a mirror image of the data resident in the PC ram. This mirror image may not necessarily reflect the exact physical shape of the data in the PC ram because of the way it is distributed across each transputer within the T-N. The creation of the PC mirror image happens at system start-up. It is important to have some sort of routines to enforce consistency between the PC and the T-N data. This is because operations in the PC and the T-N may not be synchronous.

A translator is also required for transforming the data travelling from the PC to the T-N or vice versa if they do not share the same data structures. For efficiency purposes it is highly probable for them to have completely different data structures.
Logical Diagram of the proposed Database System

Some extra components that do not appear in Brian's original sketch.

Translator - for transforming data that will compile to the interface protocol set for the PC and transputers.

Consistency checker - for keeping data in the Transputers ram and PC ram consistence. This is activated every time when an assertion or deletion happens in either the PC or transputers.

Communicator - for controlling the flow of commands and data between the PC and transputer.
Notes regarding data structures

The primary memory within the PC is not connected directly to the transputer network it therefore rule out the possibility of transferring the PC data onto the network without entailing any data transformation. The data structures vanish as soon as it leaves the memory so I wonder if any significant advantages can be gained if both systems share the same data structure. It looks like a translator is inevitable for the future hybrid system.

If a translator is to be built it will be responsible for converting data from either side into a format that compile to the protocol (to be drafted). This is used for data passing through the PC/Transputer Interface.

In my system the notion of context is not implemented. The main body of the knowledge base graph is a collection of all fully expanded, instantiated conceptual graphs maximally joined together. This is partitioned and spread over the network. Others data structures such as Types Definition, Conformity Table, Lattices are present on the master as well as in each slave. However these data structures in the slaves are differ from the master in only one respect. They only contain partial information just sufficient for dealing with the partition knowledge base graph in their storage.

Conformity Table (appeared in Master Transputer)

```cpp
class ConformTabUnit {
public:
    char * label, //Concept Label
    * ref; //Referent Label
    Node * addr; //Location in knowledge base
    Def * def; //Location in definition list
    Type * latticeAddr; //Location in lattice
    int boundary; //Flag for indicating if it is a boundary node
    int[4] location; //Location in transputer network
    ConformTabUnit * next //Linkage to next ConformTabUnit
    ConformTabUnit * prev //Linkage to previous ConformTabUnit

    ConformTabUnit constructor, destructor;
};
```

The Conformity Table stored in the master transputer contains all the details concerning Concept labels and Referent label that existed across
the whole network. Locations of concepts or relations in various local data structures can also be found using this table.

Conformity Table (appeared in Slave Transputer)

class ConformTabUnit {
public:
    char * label, //Concept Label
    * ref;       //Referent Label
    Node * addr; //Location in knowledge base
    Def * def;   //Location in definition list
    Type * latticeAddr; //Location in lattice
    ConformTabUnit* next, //Linkage to next ConformTabUnit
    * prev;  //Linkage to prev ConformTabUnit

    ConformTabUnit constructor, destructor...
}

This is the same as the Conformity Table stored in the Master except the attributes boundary and location are missing. If for any reason this missing information is needed it is obtained by issuing a request call to the Master Conformity Table.

All the data structures below are doubly linked so that modification such as addition or deletion of a single item is a bit faster.

Knowledge Base (Slave)

class KB {
public:
    Nodes * toKB;    //Pointer to KB
    KB * prev,      //Linkage to next KB
        * next;
}

This is used for organising the knowledge base graphs resident in the slave. Entry in toKB is knowledge base graph that cannot be connected to others.

1. Connections used for linking Concepts and Relations (Master & Slave)

class ptr {
public:
    Node * ToNode;   //Pointer to Node
    ptr * prev,      //Linkage to ptr
        * next;

    ptr constructor, destructor...
}

This is equivalent to Concept List Element with ToNode alias PointedAt and next alias NextInList.

2. Concept and Relation Node (Master & Slave)
class Node {
public:
    int type; //For indicating if the node is a Concept or Relation node
    int boundary; //Flag for indicating if it is a boundary type
    char * label, * ref; //Labels
    Def * def; //Pointer to its definition
    ptr * DirectedTo, * DirectedBy; //Pointers to nodes pointing at it
    Node * prev, * next; //Pointers to nodes it is pointing at

    Node constructor, destructor...
}

This is used for both the Concept and Relational types for constructing graphs used in the Definitions and KB. This is equivalent to your Concept and Relation.

3. Connections used for linking types in lattice (Master & Slave)
class ptrL {
public:
    Type * ToType; //Pointer to Type
    ptrL * prev, * next; //Linkage to ptrL

    ptrL constructor, destructor...
}

ToType and prev are alias to Pointed At and NextInList respectively in LatticeEltPtr.

4. Lattice Element (Master & Slave)
class Type {
public:
    ptrL * SuperType, * SubType; //Pointer to its SuperType
    * prev, * next; //Linkages to others Type
    Def * def; //Pointer to its definition
    char * label; //Label
    int visited; //Flags for showing if it has been visited

    Type constructor, destructor...
}

Equivalent to LatticeElt.

5. Concept and Relation Definition (Master & Slave)
class Def {
public:
char * DefLabel;          //Definition name
Node * genusA,          //Pointer to genus in the definition
    * genusB;          //Pointer to genus in the definition
Def * next,            //Pointer to next Def
    * prev;           //Pointer to prev Def

Def constructor, destructor...

Equivalent to Defn except genusA and genus B are pointed to the
genus nodes in the definition graph.
Parallel Approach to Efficient Relational Decomposition

Alfred Chan

3/8/94

1.0 Introduction

This report deals with the subject of improving the speed performance of Bowen Relational Decomposition Algorithm with the use of parallel computer namely transputers. Candidates that are considered for changes are the Ullman and Beeri algorithms. The report begins with a brief description of the transputer and pitfalls to avoid in designing a parallel algorithm based on it. This is followed by the design of the Parallel Decomposition Algorithm. Changes that are made are recapped at the end of the report together with a discussion on others' areas where changes are not applied due to uncertain factors.

2.0 Transputer

The paradigm of the transputer is MIMD (Multiple Instruction Multiple Data) where each transputer has its own private memory. Transputers can be connected to each other through its hardware links to form a network in solving problems. This is advantageous over architecture with shared memory as they always suffered from bus contention when more processing elements are added. To create a successful parallel algorithm using transputer, special attentions have to be made to localise the data in each transputer. Failure to achieve this will result in huge amount of time spent transporting data between transputers. Additional resources such as special routines have to be accommodated to take care of this inter transputer's traffic. This not only wasted resources but also increases the chance of communication deadlock making the algorithm over complex, inefficient and slow.

2.1 Strategies for Parallelisation and Limitations

Two strategies can be applied to the decomposition algorithm to take advantage of the speed increase offered by the transputers. In general, the whole algorithm can be executed in parallel (Strategy A) or alternately, parts of the algorithm can be executed in parallel (Strategy B). The feasibility of both options depends greatly on data used by the algorithm. These are the input data, the data accessed by the algorithm and the data modified by the algorithm. The nature of these data can affect the likelihood of running the algorithm in parallel. Listed below are some limitations that prevent the algorithm from gaining speed up with parallel processing.

- **The input data** - If the input data is a function of the previous output data then strategies A cannot be applied nevertheless strategy B still prevail.

  i.e.
  
  \[
  I \text{ is the set of input data} \\
  O \text{ is the set of output data} \\
  \text{where } I_0 = I_0 \text{ and } I_i = O_{i-1} \text{ for } i > 0.
  \]

- **Data accessed by the algorithm** - If the data used by the algorithm changes after each execution then many resources is needed to keep the
rest of the data residing in different transputer in accord with the rest. This is unfavourable for both strategies unless they do not make use of the modified data.

i.e.
D is the data used by the algorithm
I is the set of input data
O is the set of output data

\[
\begin{align*}
\text{begin} & \quad \text{// A Simple Algorithm} \\
O_i &= I_i \ast D; \quad \text{D} = \text{D} - I_i; \\
\text{end};
\end{align*}
\]

- Modifications made to the data - This is basically the same as the above but the modified data in this case may not be used at all for processing by the algorithm. This could be a major hindrance if the modification of data depends on its previous state. Again this is not suitable for both strategies.

i.e.
D is the data used by the algorithm
P is the set of data modified by the algorithm
I is the set of input data
O is the set of output data

\[
\begin{align*}
\text{begin} & \quad \text{// A Simple Algorithm} \\
I_i &= I_i + D; \\
P_i &= I_i ** 2 + P_i; \quad \text{// Changes in P(n)} \\
O_i &= I_i; \\
\text{end};
\end{align*}
\]

3.0 Bowen Algorithm

Embedded within the Bowen algorithm are two algorithms that are of interest to us. Since both of these algorithms possess none of the above restrictions about data, it is straightforward to transform them from their sequential version to parallel version. These are the classic Ullman and Beeri algorithms. Detail's description of these algorithms can be found in Bowen's document.

The transputer's topology adopted for supporting the theoretical parallel versions of Ullman and Beeri algorithm is based on processor farm arrangements. All transputers are linked as a pipeline, sitting at the start of the pipeline is the Master transputer. The rest of the transputers are slave to the Master. The role of the master concerns issuing tasks to the slaves, distributing data to the slaves and receiving results from the slaves.

Others' topologies can also be considered but it is not appropriate to discuss them at this early stage.

3.1 Ullman Algorithm

This is used for calculating the closure of types X concerning F where F is the functional dependency. Looking at an example given in Ullman, J. D. (1988) Principles of Database and Knowledge-Base System Volume 1 it can be seen that the nature of the data used by the algorithm
can be made in such a way that there is no violation to good parallel design theme as stated in earlier sections.

\[
U = \text{the type domains ABCDE} \\
F = \{ BC \rightarrow D, C \rightarrow A, CF \rightarrow BD, CE \rightarrow AF, AB \rightarrow C, D \rightarrow EF, ACD \rightarrow B, BE \rightarrow C \} \\
X = \text{the type domains BD}
\]

An example from Ullman

\[X^+\] is reached when the algorithm reached a state where \(X_i = X_{i+1}\). Starting with \(X_0 = BD\), we can get \(X_1\) by finding functional dependency with determinates of \(B, D,\) or \(BD\). Since only \(D\) can be matched to \(D \rightarrow EF\), \(X_1 = D \rightarrow EF\). Similarly, determinates accumulated from two previous matching are used for finding \(X_2\) and so on. This continues until the closure is reached or closure cannot be found.

**Determinates for matching next \(X\)**

\[
x_0 = BD \\
x_1 = D \rightarrow EF \\
x_2 = BE \rightarrow C \\
\]

**3.1.1 Parallel Matching of Determinates**

From the above example the matching of determinates can be processed in parallel. This can be achieved by distributing a partition of the set \(F\) into transputers where no two transputer will contains duplicate \(F\) members. This partitioned functional dependencies will stay in their destined location throughout the whole closure computation. Reload of functional dependencies arise when changes are made to \(F\). It is assumed that this will not occur during the computation of the closure. For each matching step the determinates for finding \(F\) members are send to each transputer and matches are carried out locally. Matched functional dependencies are send back to the master. For \(n\) transputer the speedup achieved can be as close as \(n\) time assuming the transferring of data is minimal.

**3.2 Beeri Algorithm**

The Beeri Algorithm is used to generate a nonloss dependency-preserving decomposition into 3NF. The main part of the algorithm comprised of partitioning a set of functional dependency into individual functional dependency and the forming of schemes based on the partitioned functional dependencies.

**3.2.1 Pipeline Partitioning of functional dependency**

The requirement for the partitioning of the set of functional dependency \(F\) is that no two groups are allowed to have the same determinate side. This is similar to sorting the set into groups with unique determinate sides. The sortings necessitate the checking of each member of the \(F\) against the whole set of \(F\) for matches. This is well suited for pipeline processing.

The set of functional dependency \(F\) is partitioned into smaller set and distributed to each slave. Members from \(F\) are fed into the pipeline from the
Master continuously one by one. Each member will pass through all the slaves and arriving back to the Master in a round robin fashion. When each member goes through a slave, it tried to find member with the same determinates sides. If matches occur it made a duplicate copy of the member and carry it along to the next slave, a tag is set for the matches to indicate these F have been matched and are taken off the F list. As soon as each member return to the Master with or without matches the formation of schemes can take place right away.

4.0 Conclusion

The prime objective of this report has been reached by showing that running Bowen algorithm in parallel is feasible. This is accomplished by executing parts of Bowen algorithm in parallel namely the Ullman and Beeri algorithms. Estimation of the effective speedup from it however cannot be made at this point due to inadequate information regarding the preparation of data for parallel processing, cost for merging results obtained from parallel processing and the communication cost for shuffling the data around the transputers. The cost estimate is not important at this preliminary stage, however. Nevertheless, this should be included in later stage otherwise there is no mean to quantify the changes from running the algorithm in parallel.

A more significant speed increase can be achieved if all assertion of new relation can be performed in parallel. Implication such as modifications made to the template by the algorithm sadly undermined this option. This situation can be rectified if changes made to temple can be postponed until all new relation are asserted.

5.0 Reference

Brian Bowen (1994), Technical Note: Efficient relational storage of conceptual graphs based on automatic extraction of dependencies.
// Translator PC->Transputer

#include "PC_Link.h"
#include <stdio.h>
#include <stdlib.h>

//Procedure for sending graph as a form of tuples to the Transputer

void Send_Graph_Tuple (PtrNode Relation)
{
    int i = 0;
    int OK = 99;                // Continuation flag
    int EOT = 0;                // End of Transmission flag
    int SendGraph = 1;          // Signaling the transputer to receive
                                // graph tuples
    PtrNode Head = Relation;

    PC_LinkByteOut (SendGraph);

    while (Head)
    {
        PC_LinkByteOut (OK);

        // Sink Concept Node
        while (Relation->DirectedBy->label[i++] != '\0')
            PC_LinkByteOut (Relation->DirectedBy->label[i]);
        i = 0;
        PC_LinkByteOut (EOT);

        // Relation Node
        while (Relation->DirectedTo->label[i++] != 0)
            PC_LinkByteOut (Relation->label[i]);
        i = 0;
        PC_LinkByteOut (EOT);

        // Sink Concept Node
        while (Relation->label[i++] != 0)
            PC_LinkByteOut (Relation->DirectedTo->label[i]);

        Head = Head->next;
    }

    PC_LinkByteOut (EOT);
}

//Transputer Procedure for receiving graph tuples

PtrNode GraphTuplesCollector ()
{
    PtrNode Head = 0;
    PtrNode SourceConcept = 0;
    PtrNode RelationNode = 0;
    PtrNode SinkConcept = 0;
    PtrLink Link = 0;
    char Concept[30];
    char Relation[30];
    int EOT = 0;
    int i = -1;

    while (ChanInInt (Host) != EOT)
    {

SourceConcept = NodeAlloc();
RelationNode = NodeAlloc();
SinkConcept = NodeAlloc();
if (Head == 0)
    Head = RelationNode;
else
{
    RelationNode->prev = Head;
    Head->next = RelationNode;
    Head = RelationNode;
}

while (Concept[i] != EOT)
{
    i++;
    Concept[i] = ChanInInt (host);
}
strcpy (SourceConcept->label,Concept);
Concept[0] = '\0'; i = -1;
while (Relation[i] != EOT)
{
    i++;
    Relation[i] = ChanInInt (hots);
}
strcpy (RelationNode->label,Relation);
Relation[0] = '\0'; i = -1;
while (Concept[i] != EOT)
{
    i++;
    Concept[i] = ChanInInt (host);
}
strcpy (SinkConcept->label,Concept);
Concept[0] = '\0'; i = -1;

SourceConcept->DirectedTo = RelationNode;
RelationNode->DirectedBy = SourceConcept;
RelationNode->DirectedTo = SinkConcept;
SinkConcept->DirectedBy = RelationNode;
}
return Head;
}

//Transputer procedure for constructing graph based on the graph tuples
void GraphConstructor (PtrNode &Head)
{
    PtrNode SourceConcept = 0;
    PtrNode SinkConcept = 0;
    PtrNode Start = Head;
    PtrNode StartTwo = Head;
    PtrNode Item = 0;
    PtrStack SourceStack = 0;
    PtrStack SinkStack = 0;
    PtrStack VisitStack = 0;

    while (Start)
    {
        if (Start->DirectedBy->visited == 0)
{ 
    SourceConcept = Start->DirectedBy;
    Start->DirectedBy->visited = 1;
    Push (VisitStack,Start->DirectedBy);
}

if (Start->DirectedTo->visited == 0)
{
    SinkConcept = Start->DirectedTo;
    Start->DirectedTo->visited = 1;
    Push (VisitStack,Start->DirectedTo);
}
while (StartTwo)
{
    if (StartTwo->DirectedBy->visited == 0)
    {
        if (strcmp(SourceConcept->label,StartTwo->DirectedBy->label) == 0)
        {
            StartTwo->DirectedBy->visited = 1;
            Push (SourceStack,StartTwo->DirectedBy);
            Push (VisitStack,StartTwo->DirectedBy);
        }
        if (StartTwo->DirectedBy->visited == 0)
        if (strcmp(SinkConcept->label,StartTwo->DirectedBy->label) == 0)
        {
            StartTwo ->DirectedBy->visited = 1;
            Push (SinkStack,StartTwo->DirectedBy);
            Push (VisitStack,StartTwo->DirectedBy);
        }
    }
    if (StartTwo->DirectedTo->visited == 0)
    {
        if (strcmp(SourceConcept->label,StartTwo->DirectedTo->label) == 0)
        {
            StartTwo->DirectedTo->visited = 1;
            Push (SourceStack,StartTwo->DirectedTo);
            Push (VisitStack,StartTwo->DirectedTo);
        }
        if (StartTwo->DirectedTo->visited == 0)
        if (strcmp(SinkConcept->label,StartTwo->DirectedTo->label) == 0)
        {
            StartTwo->DirectedTo->visited = 1;
            Push (SinkStack,StartTwo->DirectedTo);
            Push (VisitStack,StartTwo->DirectedTo);
        }
    }
    StartTwo = StartTwo->next;
}
StartTwo = Head;
while (Item = Pop (SourceStack)
    ConnectNodes (SourceConcept,Item);
while (Item = Pop (SinkStack)
    ConnectNodes (SinkConcept, Item);

    Start = Start->Two;
}

Start = Head;
Item = 0;
while (Start)
{
    Start->DirectedBy->next = Start->DirectedTo->DirectedTo;
    Start->DirectedTo->DirectedTo->prev = Start->DirectedBy;

    if (Item != 0)
    {
        Start->DirectedBy->prev = Item;
        Item->next = Start->DirectedBy;
        Item = Start->DirectedTo->DirectedTo;
        Start = Start->next;
    }
}
ClearVisits (VisitStack);
typedef struct {
    Concept *Conc;
    Node *NewNode;
} NodeMapElt;

Maps->Conc = CopyConcs;
Maps->NewNode = NewConc;
Maps->NextInList = 0;
CopyPass = NOTFIRST;

for (EndNew = NewInstance; EndNew->NextInList != 0; EndNew = EndNew->Next){
    for (AddRel = DownCList->RelationList; AddRel != 0; AddRel = AddRel->NextInList){
        AddRelation = AddRel->PointedAt;
        AddRelDir = AddRel->Direction;
        if (AddRelDir == FROM) {
            AddRelFound = AddRelation->To;
            for (EndMap = Maps; EndMap->NextInList != 0; EndMap = EndMap->NextInList){
                NewMap = typealloc(NodeMapElt);
                NewMap->Conc = CopyConcs;
                NewMap->NewNode = NewConc;
                NewMap->NextInList = 0;
                Maps = Maps->NextInList;
                NewMap = NewMap->NextInList;
            }
            for (EndMap = Maps; EndMap->NextInList != 0; EndMap = EndMap->NextInList){
                NewMap = typealloc(NodeMapElt);
                NewMap->Conc = CopyConcs;
                NewMap->NewNode = NewConc;
                NewMap->NextInList = 0;
                Maps = Maps->NextInList;
                NewMap = NewMap->NextInList;
            }
        }else {
            for (EndMap = Maps; EndMap->NextInList != 0; EndMap = EndMap->NextInList){
                NewMap = typealloc(NodeMapElt);
                NewMap->Conc = CopyConcs;
                NewMap->NewNode = NewConc;
                NewMap->NextInList = 0;
                Maps = Maps->NextInList;
                NewMap = NewMap->NextInList;
            }
        }
    }
}

for (EndMap = Maps; EndMap->NextInList != 0; EndMap = EndMap->NextInList){
    NewMap = typealloc(NodeMapElt);
    NewMap->Conc = CopyConcs;
    NewMap->NewNode = NewConc;
    NewMap->NextInList = 0;
    Maps = Maps->NextInList;
    NewMap = NewMap->NextInList;
}
else {
    AddRelFound = AddRelation->From;
}
if (AddRelFound->Used == NOTYETDEF)(
    GetOut = FALSE;
    for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList){
        if (LocMap->Conc == DownCList){
            First = LocMap;
            GetOut = TRUE;
        }
    }
    GetOut = FALSE;
    for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList){
        if (LocMap->Conc == AddRelFound){
            Second = LocMap;
            GetOut = TRUE;
        }
    }
    Map1 = First->NewNode;
    Map2 = Second->NewNode;
}

NewRel = typealloc(Node);
NewRel->Type = RELATION;
NewRel->Boundary = NOTYETDEF;
strncpy (NewRel->Label, AddRelation->Label, 25);
strncpy (NewRel->Referent, '\0', 25);
NewRel->Def = NOTYETDEF;
NewRel->DirectedTo = 0;
NewRel->DirectedFrom = 0;
NewRel->Next = 0;
for (EndList = NewInstance; EndList->Next != 0; EndList = EndList->Next){
    EndList->Next = NewRel;
    NewRel->Prev = EndList;
}

Map1Ptr = typealloc(Ptr);
Map1Ptr->ToNode = Map1;
strncpy (Map1Ptr->TableName, '\0', 25);
Map1Ptr->Prev = 0;
Map1Ptr->Next = 0;
Map2Ptr = typealloc(Ptr);
Map2Ptr->ToNode = Map2;
strncpy (Map2Ptr->TableName, '\0', 25);
Map2Ptr->Domain = 0;
Map2Ptr->Prev = 0;
Map2Ptr->Next = 0;

if (AddRelDir == FROM){
    NewRel->DirectedBy = Map2Ptr;
    NewRel->DirectedTo = Map1Ptr;
}
else {
    NewRel->DirectedBy = Map1Ptr;
    NewRel->DirectedTo = Map2Ptr;
}

NewRL = typealloc(Ptr);
NewRL->ToNode = NewRel;
strncpy (NewRL->TableName, AddRel->TableName, 25);
NewRL->Domain = AddRel->Domain;
NewRL->Prev = 0;
NewRL->Next = 0;
ReciprocalRL = typealloc(Ptr);
ReciprocalRL->PointedAt = NewRel;
ReciprocalRL->Prev = 0;
ReciprocalRL->Next = 0;
Hit = FALSE;
for (Back = AddRelFound->RelationList; Hit == FALSE; Back = Back->NextInList){
    BackRel = Back->PointedAt;
    for (EndList = BackRel->RelationList; EndList->Next != 0; EndList = EndList->Next){
        EndList->Next = NewRL;
        NewRL->Prev = EndList;
    }
}
BackDir = Back->Direction;
if (BackRel == AddRelation && AddRelDir != BackDir){
    Hit = TRUE;
    strncpy (ReciprocalRL->TableName, Back->TableName, 25);
    ReciprocalRL->Domain = Back->Domain;
    ReciprocalRL->Done = 'N';
}
if (Back->NextInList == 0)
    Hit = TRUE;

/* ...and hook them up to the required relations. */
/*******************************************************************************/
if (AddRelDir == FROM){
    if (Map1->DirectedBy == 0){
        Map1->DirectedBy = NewRL;
    } else {
        for (End = Map1->DirectedBy; End->Next != 0; End = End->Next){
            End->Next = NewRL;
        }
    }
    if (Map2->DirectedTo == 0){
        Map2->DirectedTo = ReciprocalRL;
    } else {
        for (End = Map2->DirectedTo; End->Next != 0; End = End->Next){
            End->Next = ReciprocalRL;
        }
    }
} else {
    if (Map1->DirectedTo == 0){
        Map1->DirectedTo = NewRL;
    } else {
        for (End = Map1->DirectedTo; End->Next != 0; End = End->Next){
            End->Next = NewRL;
        }
    }
    if (Map2->DirectedBy == 0){
        Map2->DirectedBy = ReciprocalRL;
    } else {
        for (End = Map2->DirectedBy; End->Next != 0; End = End->Next){
            End->Next = ReciprocalRL;
        }
    }
}
/*******************************************************************************/
// Mark off a completely processed concept. */
/*******************************************************************************/
DownCList->Used = TRUE;
}
for (DownCList == CurrConc; DownCList != 0; DownCList = DownCList->Next){
    DownCList->Used = NOTYETDEF;
}
return (NewInstance);
}

Concept *Chan_Bowen_Conversion (CurrNode)
Node *CurrNode;
{
    typedef struct {
        Concept *Conc;
        Node *NewNode;
        int Used;
        struct NodeMapElt *NextInList;
    } NodeMapElt;
    RelationListElt *Map1Ptr, *Map2Ptr, *End;
    Relation *NewRel;
    int CopyPass, GetOut, Hit, Status;
Object of this routine - to convert structures used by Alfred Chan's parallel system (for doing projection of simple graphs over transputers, etc) into the usable in the C-GRASS system (as devised by yours truly, "Bish" Bowen). The first thing done is to work through the Nodes in a Chan-structured graph and convert them into Bowen-structure Concepts; NewInstance is the head node, and will be returned at the end of the routine - it points to a linked list of Concepts.

Maps = typealloc(NodeMapElt);
CopyPass = FIRST;
for (CopyConcs = CurrNode; CopyConcs != 0; CopyConcs = CopyConcs->Next){
    if (CopyConcs->Type == CONCEPT){
        NewConc = typealloc(Concept);
        strncpy (NewConc->Label, CopyConcs->Label, 25);
        strncpy (NewConc->Ref, CopyConcs->Referent, 25);
        NewConc->LineOfIdentity = NOTYETDEF;
        NewConc->RelationList = 0;
        NewConc->GenusFlag = NOTGENUS;
        NewConc->Used = NOTYETDEF;
        NewConc->Next = 0;
        if (CopyPass == FIRST){
            NewInstance = NewConc;
            NewNode = CopyConcs;
            Maps->NewNode = NewConc;
            Maps->Conc = NewConc;
            Maps->Used = NOTYETDEF;
            Maps->NextInList = 0;
            CopyPass = NOTFIRST;
        } else {
            for (EndNew = NewInstance; EndNew->Next != 0; EndNew = EndNew->Next){
                EndNew->Next = NewConc;
                for (EndMap = Maps; EndMap->NextInList != 0; EndMap = EndMap->NextInList){
                    NewMap = typealloc(NodeMapElt);
                    NewMap->NewNode = CopyConcs;
                    NewMap->Conc = NewConc;
                    Maps->Used = NOTYETDEF;
                    NewMap->NextInList = 0;
                    EndMap->NextInList = NewMap;
                }
            }
        }
    } else {
        for (AddRel = DownCList->Next; AddRel != 0; AddRel = AddRel->Next){
            if (AddRel->ToNode == AddRelFound.Ptr){
                Status = GetMark->Used;
                Hit = TRUE;
            } else {
                for (LocMap = Maps; Hit == FALSE; LocMap = LocMap->NextInList){
                    if (LocMap->NewNode == DownCList){
                        First = LocMap;
                        GetOut = TRUE;
                    } else {
                        GetOut = FALSE;
                        for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList){
                            if (LocMap->NewNode == AddRelFound.Ptr){
                                Second = LocMap;
                                GetOut = TRUE;
                            } else {
                                GetOut = FALSE;
                                for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList){
                                    if (LocMap->NewNode == AddRelFound.Ptr){
                                        Second = LocMap;
                                        GetOut = TRUE;
                                    } else {
                                        GetOut = FALSE;
                                        for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList){
                                            if (LocMap->NewNode == AddRelFound.Ptr){
                                                Second = LocMap;
                                                GetOut = TRUE;
                                            } else {
                                                GetOut = FALSE;
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
Map1 = First->Conc;
Map2 = Second->Conc;

/* Make a relation... */

NewRel = typealloc(Relation);
strncpy (NewRel->Label, AddRelation->Label, 25);
NewRel->To = 0;
NewRel->From = 0;

/* ...and make the RelationListElts that link the relation to its concepts. */

Map1Ptr = typealloc(RelationListElt);
Map1Ptr->pointedAt = NewRel;
Map1Ptr->Direction = FROM;
strncpy (Map1Ptr->TableName, AddRelFoundPtr->TableName, 25);
Map1Ptr->Domain = AddRelFoundPtr->Domain;
Map1Ptr->Done = 'N';
Map1Ptr->NextInList = 0;
Map2Ptr = typealloc(RelationListElt);
Map2Ptr->pointedAt = NewRel;
Map2Ptr->Direction = TO;
Hit = FALSE;
for (Back = AddRelFound->DirectedTo; Hit == FALSE; Back = Back->Next){
    BackRel = Back->ToNode;
    if (BackRel == AddRelation){
        Hit = TRUE;
        strncpy (Map2Ptr->TableName, Back->TableName, 25);
        Map2Ptr->Domain = Back->Domain;
    }
    if (Back->Next == 0){
        Hit = TRUE;
    }
}
Map2Ptr->Done = 'N';
Map2Ptr->NextInList = 0;

/* Now link the made relation to its concepts... */

NewRel->From = Map2Ptr;
NewRel->To = Map1Ptr;

/* ...and hook those concepts up to the RelationListElts. */

if (Map1->RelationList == 0){
    Map1->RelationList = Map1Ptr;
} else {
    for (End = Map1->RelationList; End->NextInList != 0; End = End->NextInList){
        End->NextInList = Map1Ptr;
    }
}
if (Map2->RelationList == 0){
    Map2->RelationList = Map2Ptr;
} else {
    for (End = Map2->RelationList; End->NextInList != 0; End = End->NextInList){
        End->NextInList = Map2Ptr;
    }
}

/* Now move on to deal with this concept node's DirectedTo list. This is much the */
/* same as before, except that the list looked at is different, and the directions */
/* given to the RelationListElts is reversed. */

for (AddRel = DownCList->DirectedTo; AddRel != 0; AddRel = AddRel->Next){
    AddRelation = AddRel->ToNode;
    AddRelFoundPtr = AddRelation->DirectedTo;
    AddRelFound = AddRelFoundPtr->ToNode;
    Hit = FALSE;
    for (GetMark = Maps; Hit == FALSE; GetMark = GetMark->NextInList){
        if (GetMark->NewNode == AddRelFound){
            Status = GetMark->Used;
            Hit = TRUE;
        }
    }
}
if (Status == NOTYETDEF)

GetOut = FALSE;
for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList)

if (LocMap->NewNode == DownCList)

First = LocMap;
GetOut = TRUE;
}
GetOut = FALSE;
for (LocMap = Maps; GetOut == FALSE; LocMap = LocMap->NextInList)

if (LocMap->NewNode == AddRelFound)

Second = LocMap;
GetOut = TRUE;
}
Map1 = First->Conc;
Map2 = Second->Conc;

/* Make a relation again... */
/]***********************************************************************/
NewRel = typealloc(Relation);
strncpy (NewRel->Label, AddRelation->Label, 25);
NewRel->To = 0;
NewRel->From = 0;

/*...and make the RelationListElts that link the relation to its concepts. */
/]***********************************************************************/
Map1Ptr = typealloc(RelationListElt);
Map1Ptr->PointedAt = NewRel;
Map1Ptr->Direction = TO;
strncpy (Map1Ptr->TableName, AddRelFoundPtr->TableName, 25);
Map1Ptr->Domain = AddRelFoundPtr->Domain;
Map1Ptr->Done = 'N';
Map1Ptr->NextInList = 0;
Map2Ptr = typealloc(RelationListElt);
Map2Ptr->PointedAt = NewRel;
Map2Ptr->Direction = FROM;

Hit = FALSE;
for (Back = AddRelFound->DirectedBy; Hit == FALSE; Back = Back->Next)

if (BackRel == AddRelation)

Hit = TRUE;
strncpy (Map2Ptr->TableName, Back->TableName, 25);
Map2Ptr->Domain = Back->Domain;
}
if (Back->Next == 0)

Hit = TRUE;
}
Map2Ptr->Done = 'N';
Map2Ptr->NextInList = 0;

/* Now link the made relation to its concepts... */
/]***********************************************************************/
NewRel->From = Map2Ptr;
NewRel->To = Map1Ptr;

/*...and hook those concepts up to the RelationListElts. */
/]***********************************************************************/
if (Map1->RelationList == 0)

Map1->RelationList = Map1Ptr;
else {
for (End = Map1->RelationList; End->NextInList != 0; End = End->NextInList)

}
End->NextInList = Map1Ptr;
if (Map2->RelationList == 0)

Map2->RelationList = Map2Ptr;
else {
for (End = Map2->RelationList; End->NextInList != 0; End = End->NextInList)

}
End->NextInList = Map2Ptr;
/** Mark off a completely processed concept. */

for (LocMap = Maps; LocMap != 0; LocMap = LocMap->Next){
    if (LocMap->NewNode == CurNode){
        LocMap->Used = TRUE;
    }
}

return (NewInstance);
This appendix contains a reprint of [Bowen and Kocura 1993], which was presented to the First International Conference on Conceptual Structures in Quebec City. It is of note mainly from a historical angle, as it shows the early use of a binary relational strategy by C-GRASS.
Implementing Conceptual Graphs in a RDBMS

Brian Bowen and Pavel Kocura
B.A.Bowen@uk.ac.loughborough, P.Kocura@uk.ac.loughborough

Department of Computer Studies
Loughborough University
Loughborough, Leics LE11 3TU

Abstract: This paper discusses why the relational data model is inadequate for the modelling of certain data domains, and proposes that conceptual graphs can be used to remedy some of these inadequacies. A description of current research into implementing a conceptual graph layer on top of a relational database is given, along with some general algorithms for conceptual graph operations on relational database structures.

Keywords: Relational Databases, Graph Storage, Graph Retrieval, Data Constraints

1 Introduction

Since its introduction by Codd in 1970, the relational model has dominated the database world. Systems built on this model offer a consistent, integrated way to manage large amounts of data, and yet have a mathematically sound and conceptually simple basis. Their success stems primarily from the fact that a simple set of operations can process large volumes of data efficiently.

However, it is increasingly obvious that this model has been difficult to adapt to a sizable and growing number of data modelling domains that have arisen over the past few years. Such domains share some common features:

* they all contain complex objects that are very difficult to represent in a relational database (hereafter referred to as RDB);

* they require more manipulative power than the relational model can provide. Such manipulations are often closely linked with the object being modelled;

* such domains often require the modelling of complicated interrelationships and constraints associated with the objects being modelled; the constraint mechanisms of RDBs are completely unable to cope with such requirements.

A good example of a simple application that is badly served by the relational model is given by Stonebraker [1989], who supposes that a user wants to store the layout of
Manhattan; the application models data in the form of two-dimensional boxes, which are represented by the co-ordinates of their two corner points, \((X_1, Y_1)\) and \((X_2, Y_2)\). If we try to store this in an RDB, a reasonable approach would be the BOX relation:

<table>
<thead>
<tr>
<th>IDENT</th>
<th>(X_1)</th>
<th>(Y_1)</th>
<th>(X_2)</th>
<th>(Y_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
</tbody>
</table>

Now, suppose that we want to find all boxes in a given region defined by the corners \((0,0)\) and \((1,1)\). A relational language like SQL would query:

```
SELECT * FROM BOX
WHERE X1 <= 1 AND X2 >= 0 AND Y1 <= 1 AND Y2 >= 0
```

Even this will require, on average, half of the index records in the index to be read – no matter how they are organised\(^1\). If we have a million boxes, then 500,000 index records will need to be read on average, ensuring bad performance even on large machines. If we allowed boxes within boxes, the relational model would be overwhelmed by the complexities – whereas object-orientation would be relatively easy to adapt to such a change by modelling BOX as a data type in its own right, and assigning methods to it that could calculate such queries efficiently.

This paper proposes that conceptual graph theory [Sowa 1976, 1984] can be used to remedy some of the weaknesses in RDBs shown above. But why should we? Whilst in agreement with the view that object-oriented databases are more powerful than RDBs in terms of modelling flexibility:

* from the point of view of the RDB user, object-oriented databases are unattractive – they are essentially object-oriented programming languages with persistent pointers; their imperative, procedural nature tends to discourage users who have become accustomed to declarative Data Manipulation Languages (DMLs), and who balk at the complexity of the new approach\(^2\). So while object-oriented databases are very expressive, they lose much in terms of user-friendliness and usability.

---

\(^1\) Most common indexing mechanisms (such as B-trees and hash indexes) are unable to evade this problem.

\(^2\) This undesirable state of affairs has been recognised by some designers – for example, the ODE object-oriented database [Agrawal and Gehani 1989] possesses a pseudo-SQL interface called CQL++.
- RDBs are mathematically sound and technically robust, having much development work behind them; the same cannot be said of object-oriented databases, which are comparatively untried and do not appear to have a universally agreed basis [Stonebraker 1989].

- RDBs are the most widespread and popular form of data storage at present, and are likely to remain so for a considerable period of time to come. Even if systems based on the object-oriented model eventually establish themselves, there will still be a large number of applications where RDBs are perfectly adequate.

As regards earlier work in the area of linking conceptual graphs (hereafter referred to as CGs) and databases, Poesio [1987] has built CGs on top of an RDB (INGRES); Hines et al [1990] have demonstrated certain similarities between CGs and the object-oriented paradigm, which could form the basis of a CG–using object-oriented database; and Wuwongse and Ghosh [1992] have attempted to outline a graph–using database system based on objects and actors. Our approach tends to follow that of Poesio, mainly for the reasons mentioned earlier.

This paper is split into three sections: firstly, we explain some of the reasons for the abovementioned weaknesses in more detail, and explain how adding an extra CG layer can reduce their effect; secondly, we outline the general philosophy behind our work, and give some details of current work; finally, we give brief consideration to the future direction of our work.

2 Relational Databases and Conceptual Graphs

In this section, we amplify some of the points outlined in the introduction. What we want to know is: what are the main reasons that lie behind the weaknesses of the relational model? How can CGs be used to remedy them?

- **Modelling of complex objects**: the creation of RDB domains are limited to a small set of pre-defined types such as INTEGER, CHAR, BOOLEAN, and so on; there is no facility for the user to define a new type in terms of these primitives, or to use such types as domains in other tables.

The addition of a CG layer to an RDB replaces this limited system with domain definition via a formal system based on lambda calculus. This allows the creation of arbitrarily complex "user-defined" types which can be used as concepts (domains) in other type definitions.

- **Manipulative power**: RDBs have traditionally separated their DML from their host language (HL). The DML is primarily concerned with efficient

\[3\] Although earlier "extended relational" systems such as RM/7 [Codd 1979] have attempted to do something like this.
querying and maintenance of the database, but has little expressive power; the HL is a conventional programming language in which DML commands are normally embedded, and which is used for everything except the actual querying and maintenance of the database – it therefore has the expressive power which the DML lacks [Ulman 1988]. This separation becomes important in complex applications such as CAD/CAM and VLSI design databases, where there is a need for specialised operations as well as fast access – the standard operations of the relational model just don’t have the expressive power required, whereas object-oriented databases are essentially persistent programming languages which behave as a merged DML/HL and therefore have no problem in performing such manipulations.

The addition of a CO layer to an RDB provides a more flexible set of operations than the regular DMLs found in RDBs, and also provides an inferential component that is not found in RDBs. Computation can be performed by actors which are attached to concepts and which fire upon instantiation†; the dataflow graphs behind these structures have much of the expressive power – including that of recursive calculation – that is normally present in the HL of an RDB, but are more closely integrated with the manipulation language.

* modelling of interrelations and constraints: the explanation for this weakness comes in two parts.

Firstly, for an RDB to have any meaningful level of semantic information, there has to be some way of representing how the objects being modelled relate to each other; this is not directly possible in the relational model, although it is possible (to a very limited extent) in design formalisms such as entity-relationship modelling. The addition of a CO layer to an RDB represents these interrelations by the use of conceptual relations between concepts, which represent the domain roles that those concepts play vis-à-vis each other.

Secondly, RDBs have virtually no constraint mechanisms, apart from those explicitly declared by the programmer. This is a serious weakness, because constraints can be used to block the storage of data which violates them. The problem with programmer–defined constraints is that (a) they tend to get buried deep in the application and are therefore difficult to change, and (b) all occurrences of a given constraint must be consistent – failure to ensure this could result in one program allowing data that others would reject. The addition of a CO layer to an RDB introduces the notion of canonicity; this allows the user to represent constraints centrally, making them easy to update. The layer can also check for logical consistency (Jeffrey et al [1989] contains a useful treatment of such issues from the RDB point of view).

† Similar to some object-oriented databases, which possess “triggers”.
3 General Philosophy and Current Work

In this section, we look at some basic ideas underlying our work and how they have been put into practice.

3.1 A Few Departures From Standard Theory

Our approach to the development of our system is incremental in nature; therefore, we have initially chosen to implement only a subset of the CG theory. The most important departures from the standard theory expressed in [Sowa 1984] are as follows:

* we are restricting the graphs representable to simple dyadic connected graphs, although we do allow negated types and relations; the organisation of the system is flexible enough to allow the full representation of contexts in a more advanced system. For the time being, we merely store the negated types and relations in Peirce form, and this presents very few problems;

* although the role of inference is limited in such a restricted system, the underlying organisation of our approach – as we shall see – means that inference can be added later with no real penalty. For the time being, we are interested in maintaining the logical consistency of data, so a simple truth maintenance system is being implemented (see section 3.4 for details);

* in our basic system, there is no explicit use of sets, lambda abstractions, actors, prototypes, schemata, or composite individuals – although in a more advanced system, there would certainly be roles for the first three;

* we draft in an idea from database theory and allow the use of primitive database types such as DATE and INTEGER as specialised individual markers with specialised formats – for example, we allow [DATE: 20-08-1966] and [INTEGER: 26].

3.2 Graph Storage And Relational Databases

The way in which CGs are physically stored is of obvious importance in terms of their efficiency of retrieval, as well as of the efficiency of operations upon them. CGs can consist of many concepts and relations and are variable in size, which creates problems when attempting to store them in fixed-field relational tables. It is fairly obvious that it is necessary to fragment the stored graphs in some way. But how?

Essentially, there are two reasonable approaches to graph storage: either (i) to include extra information in the graphs (such as functional dependencies) that will allow efficient storage structures to be derived, or (ii) to fragment these irregular structures into regular substructures. Although the first option is intuitively more appealing from the point of view of algorithmic complexity, it burdens the user with an
additional workload that is absent from the second approach. After some deliberation, we have chosen to pursue the second option - using one table to store the concepts and their markers, and another to store the relation links between those markers; although this approach leads to a certain amount of data redundancy (the markers often appear more than once), this is minimal at worst; efficient indexing allows us to speedily reconstitute the graphs stored in this way quite easily. As the next section shows, this bi-tabular philosophy is applied throughout our system.

3.3 Conceptual Graphs and Storage Tables

All graph-storing tables in our system are bi-table, and have a number of domains in common. In the concept tables, the domains common to all uses are:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>A type label</td>
</tr>
<tr>
<td>INDVMARK</td>
<td>A marker, either generic or individual</td>
</tr>
<tr>
<td>ENCLOSURE</td>
<td>A simple Odd or Even</td>
</tr>
</tbody>
</table>

In the relation tables, the common domains are:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROMPOS</td>
<td>The marker in the From position⁵</td>
</tr>
<tr>
<td>RELATION</td>
<td>A relation label</td>
</tr>
<tr>
<td>TOPOS</td>
<td>The marker in the To position</td>
</tr>
<tr>
<td>ENCLOSURE</td>
<td>Again, a simple Odd or Even</td>
</tr>
</tbody>
</table>

These core domains allow us to store any CG, subject to the restrictions we outlined in section 3.1 – as an example, consider the graph:

⁵ When we mention the From and To positions, these refer to the concepts attached to a relation; the From position is occupied by a concept which points to the relation, and the To position is occupied by a concept which the relation points to.
which would be fragmented into:

<table>
<thead>
<tr>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAN</td>
<td>Brian</td>
<td>Even</td>
</tr>
<tr>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>PERSON</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>BEAUTIFUL</td>
<td>*1</td>
<td>Even</td>
</tr>
<tr>
<td>INTELLIGENT</td>
<td>*2</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian</td>
<td>CHILD.OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>CHILD.OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>ATTR *1</td>
<td>Even</td>
<td></td>
</tr>
<tr>
<td>Petra</td>
<td>ATTR *2</td>
<td>Even</td>
<td></td>
</tr>
</tbody>
</table>

where the numbered asterisks represent internal markers assigned by the system in order to ensure that the graph is logically connected when fragmented. These core domains constitute the GRAPHS database, which is used to store general information; most of the tables in our system also have extra domains that allow them to store extra information that is necessary for the data that is being stored:

* TYPECANON, which stores the positive and negative type canonical catalogues. Each tuple requires an extra domain for recording the type to which the entry belongs;

* RELATIONCANON, which stores the positive and negative relation canonical catalogues. Each tuple has extra domains for the recording of cardinality and relational properties (see section 3.4 for an outline of canonicity in this system), and for recording which relation the entry belongs to;

* TYPEDEFN and RELDEFN, which store type and relation definitions respectively. Each tuple has an extra domain to record the definition that the entry is part of.

Apart from these, we also have a number of tables which act as "accelerating" structures that speed up processing, and which are not based on the bi-tabular philosophy:

* TYPELATTICE, which stores the type lattice. This is a simple 3-ary table holding types and their subtypes, and the level of the type in the lattice;

* By this, we mean that the information stored in these tables could be extracted from the other tables — for instance, conformities could be extracted from GRAPHS; however, we allow some controlled redundancy and store them again, in order to speed up the access to this useful information.
* RELATIONLATTICE, which stores the relation lattice. Like TYPELATTICE, it is a 3-ary table and holds details of relations, their subrelations, and the relation's level in the lattice.

Our approach to storage (ie: using a number of general-purpose tables instead of the usual relational approach, which ties tables to a particular data modelling domain) means that we can, to a large extent, dispense with the need for entity-relationship modelling - because the design stage is shifted to the creation of definitions and canonical entries, which have more power than anything offered by entity-relationship modelling.

### 3.4 Canonical Constraints and Data Consistency

As well as the usual canonical features, we also add two extra constraint features - cardinality of relations and relational properties.

From a database point of view, cardinality is a natural extension of the constraint mechanisms found in CGs; the idea comes from entity-relationship modelling, which has certain similarities with CG theory but is much simpler. It provides another check on consistency by limiting the number of connections that a relation can make for any given individual marker. Initially, we have restricted our approach to the simple cardinalities - 1-1, 1-N, and so on. A simple example is the (SPOUSE) relation, which is 1-1; any attempts to involve an individual in more than one marriage violates the cardinality of the relation and is rejected. Our eventual aim in this area is to model more interesting cardinalities such as 2-5, 7-N, etc. The relational properties of reflexivity, transitivity, symmetry and antisymmetry are also useful in making sure that implausible data isn't stored. A simple example to illustrate the use of such properties is the attempted storage of a cycle of descendant (DESC) relationships:

![Diagram](image)

(DESC) is transitive, so we can derive:

![Diagram](image)

from the cycle. As (DESC) is irreflexive, this property is violated and the graph is
rejected. Apart from these two extra constraint mechanisms, we also employ a mechanism to preserve the truth consistency of the data stored. Because we are using a simplified form of graph representation, our consistency testing in our basic system is relatively straightforward.

In our basic system, consistency can be violated in two ways:

* when we have stored (1):

```
C2: #1
```

and try to store (2):

```
~C2: #1
```

or vice versa; if we say that type C1 is the immediate supertype of C2, attempting to assert (1) into a database that already contains (2) fails - because (2) is stored in the Peirce form (3):

```
C2: #1
```

(1) can therefore be projected into an opposing context in (3), which indicates failure;

* when we have stored (4):

```
C: #1
R
C: #2
```

and try to store (5):

```
C: #1
~R
C: #2
```
or vice versa: attempting to assert (4) into a database that already contains (5) fails – because (5) is stored in the Peirce form (6):

\[ C: #1 \xrightarrow{R} C: #2 \]

(4) can therefore be projected into an opposing context in (6), which indicates failure.

The addition of such constraint and consistency mechanisms to standard RDBs provides them with a level of consistency that is far superior to their more usual constraints.

3.5 Indexing and Generalisation

Having outlined the physical storage of graphs, we must now consider their logical storage - which has direct consequences for data retrieval and querying. Recently, approaches based upon the use of generalisation hierarchies for retrieval (for example, see Ellis [1989, 1990, 1991]) have emerged as a prominent method of graph storage. Essentially, such approaches store graphs in order of increasing specialisation. Notwithstanding the general acceptance of this approach, we consider it to be unsuited to approaches based on relational databases for the following reasons:

* the generalisation approach is based on a hierarchy of disjoint graphs (for example, see Ellis [1991], p. 10). This leads to widespread data redundancy on a scale not found in non-generalisation approaches. It seems to the authors that it is better to store only the most specialised graph - after all, it contains all the information found in its generalisations. Although it seems likely that the main reason for organising the data in this way is for reasons of querying efficiency, serious questions must be raised about the explosion in storage space required to achieve this. As an example, consider the storage requirements of the graph [CAT]<-[AGNT]<-[EAT] in Ellis [1990], which is based on small hierarchies; if
the hierarchies were to be made more complex, the storage requirements for even the most simple graphs would be truly phenomenal.

* deletion from a hierarchically organised system would seem be more complex than in a system based on more conventional storage (we say seem because this issue does not seem to have been addressed in any detail). Whereas our more conventional approach would clip out a single graph (and restructure a graph split into two if necessary), the hierarchical approach would probably need to rewrite many links within the hierarchy.

Therefore, we have decided not to implement any storage techniques based on generalisation, and have opted to use the standard indexing of RDBs.

3.6 Conceptual Graph Operations and DMLs

The organisation of our system leads to a number of processing problems — the usual algorithms for the implementation of graph operations such as projection and join are normally applicable only to “walkable” structures such as pointers and other such structures. We have therefore had to develop a number of new algorithms that take account of the fact that the graphs are fragmented in the way we have previously described.

Although CG operations do have certain similarities to some standard RDB operations, the extra requirements for the checking of canonicity and consistency have made the development of the new algorithms more problematic than might have been expected. Nevertheless, algorithms for the performance of three operations — COPY, PROJECTION, and SCEPTICAL JOIN — have been devised; below, we give a general description of the algorithms and a number of illustrative examples.

We consider these three operations to be the only explicit CG operations needed in our approach. This doesn’t mean that the various operations of the CG theory are absent; rather, it means that we have chosen to encapsulate the simpler operations inside a smaller number of more powerful operations. But apart from these graph manipulators, we also need to develop other operations. Although CGs subsume many of the definitional and manipulative features of RDBs, a relational DML can either be embedded in a host language (such as C) or has its own sublanguage, of which the “true” DML forms a part. We feel that a graph-based approach also requires such facilities, and we are currently engaged upon the task of building a sublanguage around our graph operations. We envisage that “programs” constructed from such a sublanguage will be based on a second-order CG, but this idea needs more development before it becomes seriously workable. As an interim step, an SQL-esque approach similar to SQUIRREL (Waugh [1989]) is also under consideration.
3.7 Graph Operations On Relational Tables – Some Examples

In this section, we outline the algorithms behind the three extended operations mentioned in section 3.6, and give a few examples of their use. Before we tackle the algorithms, let us first outline a small system that we will use to illustrate them.

The hierarchies for the examples are:

Graph #1 is:

Graph #2 is:
Graph #3 is:

We now proceed to the algorithms. The first operation is Copy, which is essentially the old canonical formation rule, but still very useful in a database context. The algorithm is trivial – just copy one table to another table – which is easily accomplished in SQL, and requires no example because it is so trivial.

The second operation to be outlined here is that of Projection. This is based on the standard operation from [Sowa 1984], and its algorithm is as follows:

IF Projector Graph is in data table form THEN
  Convert it to a pointer-based form for processing;
ENDIF
Walk pointer-based form of Projector and convert to MFs and DFs (see below);
FOR each MF DO
  Find all conformities in the Projectee data table that are projectable into by the MF
  (in terms of Type, Marker, and Enclosure);
  Store these conformities in a data table for the present - each set of conformities is
given a unique index number;
ENDDO
FOR each DF DO
  Find all dyadic relations in the Projectee data table that are projectable into by the DF
  (in terms of Relation, Markers, and Enclosure);
  Store these dyads in a data table for the present - each set of dyads is given a unique
  index number;
  Remove any dyads and conformities from indices used in earlier DF use;
ENDDO
Reorganise the dyad indices into connected graphs, in order to distinguish between
multiple projections;

A few explanatory notes are necessary at this point:

(i) MFs ("Monad Finds") and DFs ("Dyad Finds") are essentially pointer-based representations of database tuples that we wish to locate; the DF also contains extra information on how such tuples are joinable to other tuples in other DFs. Instead of being linked to single markers, the relation in a DF is linked to sets of markers - the index numbers of the MFs, which is why the MFs are all calculated prior to the DFs.

(ii) as mentioned, MFs and DFs have unique indices - primarily to distinguish similar-
Looking concepts (or whatever) from each other (for example, the node [MA] might appear at more than one place in a given graph). Reduction of redundancies relies heavily in these indices; because each DF is linked to other DFs (in the same way that relations can share common concepts which the both leave or point towards), it follows that if we have just computed the valid dyads for a DF with index X+1, then there is a possibility that some earlier (and smaller) projections are no longer valid; these must be eliminated, and we use the indices to do this.

As an example of this operation, suppose that we want to project Graph #3 into Graph #1. The table for Graph #1 (the Projectee) and the table for Graph #3 (the Projector) are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAN</td>
<td>Brian</td>
<td>Even</td>
</tr>
<tr>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>PERSON</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>BEAUTIFUL</td>
<td>'1</td>
<td>Even</td>
</tr>
<tr>
<td>INTELLIGENT</td>
<td>'2</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>'3</td>
<td>Even</td>
</tr>
<tr>
<td>PERSON</td>
<td>'4</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian</td>
<td>CHILD.OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>CHILD.OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>ATTR</td>
<td>'1</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>ATTR</td>
<td>'2</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petra</td>
<td>ATTR</td>
<td>'3</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>CHILD.OF</td>
<td>'4</td>
<td>Even</td>
</tr>
</tbody>
</table>

Because both graphs are tables, the first step is to convert the Projector to a pointer-based form. We then walk this structure to produce the following lists of MFs and DFs:

<table>
<thead>
<tr>
<th>Indx</th>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>1</td>
<td>ATTRIBUTE</td>
<td>-</td>
<td>Even</td>
</tr>
<tr>
<td>2</td>
<td>PERSON</td>
<td>-</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indx</th>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>ATTR</td>
<td>1</td>
<td>Even</td>
<td>N1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>CHILD.OF</td>
<td>2</td>
<td>Even</td>
<td>FF</td>
</tr>
</tbody>
</table>

A graph which has two dyads might successfully appear as a projection early on, but will definitely be rejected if there is a third dyad in the Projector graph - because it will no longer have the information required to be a valid projection.
In the above, the Join information in the DFs is either NJ (No Join - a special case for the first DF in the list) or FF (join the tuples found with this DF to the tuples found with the previous DF; on the From marker index position).

The next step is to locate conformities using the MFs; we get the table:

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>1</td>
<td>BEAUTIFUL</td>
<td>*1</td>
<td>Even</td>
</tr>
<tr>
<td>1</td>
<td>INTELLIGENT</td>
<td>*2</td>
<td>Even</td>
</tr>
<tr>
<td>2</td>
<td>MAN</td>
<td>Brian</td>
<td>Even</td>
</tr>
<tr>
<td>2</td>
<td>PERSON</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>2</td>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
</tbody>
</table>

We now work down the list of DFs; the first DF produces:

<table>
<thead>
<tr>
<th>Index</th>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Petra</td>
<td>ATTR</td>
<td>*1</td>
<td>Even</td>
</tr>
<tr>
<td>0</td>
<td>Petra</td>
<td>ATTR</td>
<td>*2</td>
<td>Even</td>
</tr>
</tbody>
</table>

As the index is zero, we can make an assumption that it is the first DF to be executed and so has nothing to join to (and so no redundancies to look for). The next DF in the list gives:

<table>
<thead>
<tr>
<th>Index</th>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Petra</td>
<td>CHILD OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
</tbody>
</table>

Because the From Index (0) only contains the marker Petra (see the tuples found under the MF searches), the above tuple is the only one that is valid. As the index of this DF is non-zero, we can assume that some DFs have been processed previously and attempt to remove redundancies. There are no redundancies in this case, so the final result of the projection is:
After reorganisation, we have the projections:

The last operation is that of Sceptical Join, which is a variant of the maximal join that rejects all uncanonical results. The algorithm takes two tables as input, and relies on one of the graphs being a pointer-based structure:

Convert FirstGraph into a pointer structure - SecondGraph remains as a table;
Walk pointer-based form of FirstGraph and convert to lists of MFs and DFs
(as with Projection);
FOR each MF DO
  Find all conformities in SecondGraph that have the same type label as the MF;
  Store these conformities in a data table for the present - each set of conformities is
given a unique index number;
ENDDO
FOR each DF DO
  Find all dyads in SecondGraph that have the same relation as the DF, and which have
  makers found in the DFs From and To indices;
  Remove redundancies - condition is not as stringent as in Projection;
ENDDO

* This is as in Sowa [1984].
Reorganise the dyad indices into connected graphs within join groups - each join group contains the joins possible at an interface between the two graphs, and there may be many such join sites.

Calculate the powerset of the join groups and test the canonicity of the graph resulting from the joins in the set element being tested - we start with the element containing all the local joins, and work downwards to single joins; if we discover a canonical join, we can ignore the rest and finish.

The removal of redundancies is not as stringent as in Projection, because not all parts of the graph have to interconnect - if we find a dyad which no join to the graph being joined to, it could easily be the edge of a local join, and this is the way it is treated.

As an example of this operation, suppose that we attempt to join Graphs #1 and #2. Because the canonical catalogues would contain the information that a person has a man and a woman (one of each) as parents, this join would be expected to fail. The table for Graph #1 can be found with the algorithm for Projection, and the table for Graph #2 is as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAN</td>
<td>Andrew</td>
<td>Even</td>
</tr>
<tr>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>MAN</td>
<td>Clovis</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew</td>
<td>CHILD OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>Petra</td>
<td>CHILD OF</td>
<td>Clovis</td>
<td>Even</td>
</tr>
</tbody>
</table>

The first step is to convert Graph #1 to MFs and DFs:

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Indv Mark</th>
<th>Enclosure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MAN</td>
<td>Brian</td>
<td>Even</td>
</tr>
<tr>
<td>1</td>
<td>WOMAN</td>
<td>Petra</td>
<td>Even</td>
</tr>
<tr>
<td>2</td>
<td>PERSON</td>
<td>Clovis</td>
<td>Even</td>
</tr>
<tr>
<td>3</td>
<td>BEAUTIFUL</td>
<td>*</td>
<td>Even</td>
</tr>
<tr>
<td>4</td>
<td>INTELLIGENT</td>
<td>*</td>
<td>Even</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>From</th>
<th>Relation</th>
<th>To</th>
<th>Enclosure</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>CHILD OF</td>
<td>2</td>
<td>Even</td>
<td>NJ</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>CHILD OF</td>
<td>2</td>
<td>Even</td>
<td>TT</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>ATTR</td>
<td>3</td>
<td>Even</td>
<td>FF</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>ATTR</td>
<td>4</td>
<td>Even</td>
<td>FF</td>
</tr>
</tbody>
</table>

(where TT is like FF in the Projection example, only joining on the To marker index position). We then work through the MFs to get:
which is basically identical to the MFs. As we work through the DFs, we can see that, because the conformities found by the MFs are identical to the MFs, the only DF resulting will be the second dyad of Graph #3. After reorganisation of the tables, we find that the most maximal join will give the graph:

which violates the cardinality of the CHILD_OF relation (Clovis now has three parents!); if there was no such condition of cardinality, then the join would succeed.

### 4 Future Directions

The main thrust of our future work is to exhaustively test all the parts that make up our basic system. Once this is established and working, we aim to extend it to take in all the parts of the CG theory that are useful in a database context. This would include
the addition of a more powerful system of inference (in order to infer new data which can then be stored), and the extension of our restricted graphs to the full graph formalism.

5 References


Appendix E

[Bowen and Kocura 1994]

This appendix contains a reprint of [Bowen and Kocura 1994], which was accepted for presentation to the Second International Conference on Conceptual Structures in Washington DC, but was later withdrawn by the first author on the grounds of non-attendance (although the second author was able to present a summary of the paper's main points).
Adaptive Relational Storage for Conceptual Graphs

Brian Bowen and Pavel Kocura
Department of Computer Studies
Loughborough University
Loughborough, Leicestershire
United Kingdom

Abstract: Although the relational model of data storage is considerably more declarative in its manipulation of data than earlier models, it still contains many pitfalls for the inexperienced user. The universal relation (UR) data model has been proposed as a way of increasing the declarativeness of relational query languages; we briefly examine a number of systems based on the UR approach and argue that, although such systems do increase declarativeness, their relational-like data manipulation languages restrict the effectiveness of such increases. We propose to abandon such languages in favour of the more expressive and flexible syntax of conceptual graph theory, and go on to describe a system which merges ideas from both areas; we then suggest how the underlying storage structures of the relational model can be made to adapt with reference to the data being asserted, in order to provide the user with graph retrievals that are cheaper than those found in more naive database-logic interfaces.

Keywords: Relational Databases, Universal Relations.

1. Introduction

The great advantage of the relational model over previous data models is that it has a simple yet powerful method of file navigation; this is in marked contrast to earlier models\(^1\), and is due in large part to the structural simplicity of the relational model. In comparison to earlier data models, the relational model is more declarative; by this, we mean that its data manipulation language (or DML for short) allows the user merely to "...specify only what data is wanted, not a procedure for obtaining that data" (Date [1982]). Therefore, it follows that the more declarative a DML is, the less need there is

\(^1\) Such as the network and hierarchical data models.
for application-specific navigational information required by the DML to find the data specified in the query.

1.1 Levels of Declarativeness

Although the DML of the relational model is one of the most declarative computer languages currently in common use, it still needs certain navigational information. For example, consider the following QUEL query for extracting the names and salaries of those managers earning £50,000 or more:

```
RANGE OF M IS MANAGER
RETRIEVE (M.NAME, M.SALARY)
WHERE M.SALARY >= 50000
```

We can immediately note three problems with such a query:

- the location of the data must be specified: the user needs to know the logical location of the data domains involved in the query (in our example, both NAME and SALARY can be found in the MANAGER relation).

- relation and domain names must be known: the user needs to know the exact names of the data domains and relation schemes involved in the query (in our example, NAME and SALARY are recognised domain names, and MANAGER is a recognised relation scheme name).

- syntax is very rigid: like programming languages, SQL and QUEL have rigid syntaxes that a query must adhere to. They don't allow any margin for error.

Without this knowledge, novice users are lucky if they are able to formulate even the simplest of queries, and so the novice user is constrained by a lack of knowledge about the internal representation of data; this is an unsatisfactory state of affairs because such knowledge has nothing to do with the real world of the user, and should have no bearing on the user's ability to formulate a query.

1.2 Universal Relations

It should therefore not be too surprising that quite a few attempts have been made to increase the declarativeness of relational DMLs; the most promising attempts to date seem to be ones based on the universal relation (or UR for short) data models - for
example, System/U (Ullman [1989]) and URSA (Chang and Sciore [1992])—which focus primarily on reducing the location requirement by attempting to provide complete access-path independence. The key concept behind these systems is that all underlying data structures are presented to the user in terms of a single unified table (the universal relation) which contains all domains and data found in the underlying structures. In reality, the universal relation is no more than a conceptual entity that exists at the interface level; the advantage is that the user no longer needs to know which data domain belongs to which data table, because the underlying data tables can all be seen as part of one global table; this means that the requirement to know the location of data domains largely disappears.

As a very simple example of the difference between relational and UR queries, consider the following database scheme of an investment house:

\[
\begin{align*}
OB & \ (OFFICE, BROKER#) \\
IB & \ (INVESTOR#, BROKER#) \\
SD & \ (STOCK, DIVIDEND) \\
SIQ & \ (STOCK, INVESTOR#, QUANTITY)
\end{align*}
\]

and the query: "which investors do business with Broker #100, what stocks do they have, and how many of each?"

In QUEL, the query would be fairly simple:

\[
\begin{align*}
\text{RANGE OF FIRST IS SIQ} \\
\text{RANGE OF SECOND IS IB} \\
\text{RETRIEVE (FIRST.INVESTOR#, FIRST.STOCK, FIRST.QUANTITY)} \\
\text{WHERE SECOND.BROKER#} = 100 \\
\text{AND SECOND.INVESTOR#} = \text{FIRST.INVESTOR#}
\end{align*}
\]

The universal version of this query (which we express in the syntax of System/U) is considerably simpler:

\[
\begin{align*}
\text{RETRIEVE (INVESTOR#, STOCK, QUANTITY)} \\
\text{WHERE BROKER#} = 100
\end{align*}
\]

2 Note that the term "universal relation" should not be confused with the most general or universal relation of conceptual graph theory.
The System/U query is much simpler because some of the "hard" details like the joining and table names are hidden - the details of what operations to perform are made by the system, not the user. The abolition of the requirement to know where data is stored is a key step in increasing language declarativeness, because it moves the user away from the internal data representation and closer to the real world. This is in accord with (Sowa [1984]), which gives an example of two natural language queries:

"From the skills inventory, get me the name, employee number, department, and years in service of the engineers with knowledge of German located in the New York area."

"List name, employee no., dept., and service years for engineers in New York who know German."

These two queries are rather like the QUEL and System/U queries, in that the second is obviously more declarative than the first. The second "...omits the file name, which is part of the implementation, not part of the user's view of the world" (Sowa [1984]).

While UR systems are definitely an improvement on the standard relational DMLs available, they are "...not meant to replace the relational model but rather to supplement it" (Vardi [1988]). Given the underlying structural simplicity of the model, this is unsurprising; however, the UR DMLs still have rather rigid syntax, and also need to know the domain names. In order to reduce or remove these requirements, we now suggest an approach that uses conceptual graphs as a replacement UR DML.

1.3 Adaptation of the Universal Relation Approach

As has been observed on a number of occasions (Ullman [1989], Vardi [1988]), the most declarative manipulation language is natural language. As general-purpose natural language interpreters are not yet available, we must seek an intermediate - if less powerful - solution. Perhaps the most tractable approach is to use first-order predicate logic (FOPL) as this intermediate solution; after all, there is a well-understood relationship between function-free FOPL and the relational model, and FOPL has an expressiveness that comes close to that of natural language.

3 The scope of this paper doesn't allow us to go into great detail here on how this is performed - the interested user is directed to Vardi's excellent introduction (Vardi [1988]), and Ullman's explanation of the ideas behind System/U (Ullman [1989]).
We suggest that conceptual graph theory (Sowa [1976, 1984]) can be used as a DML for a relational database manipulating data along UR lines. Conceptual graphs are equivalent to FOPL, and their theorem proof mechanism is related to the existential graph logic of Peirce\(^4\) - compared to systems that are based on Prolog, the logical facilities of a Peirce-based database system should be capable of more general modelling of logical rules. However, while conceptual graphs are capable of far greater flexibility and manipulative power than the UR DMLs, they are also rather more complex; therefore, any UR-style system which attempts to use conceptual graphs instead of the usual UR DMLs will necessarily be rather different to schemes like System/U and URSA.

2. Theoretical Background

The layout of this section is as follows: we shall first describe the basic structural ideas behind the approach - specifically the concepts of repository scheme, named scheme, and λ-scheme - and then elaborate on these basic storage ideas by discussing the rules and manipulation operations that we deem necessary for the efficient maintenance of those structures; we shall then show how these structures and operations can be used to effect the storage of a knowledge-base\(^5\) which changes its storage needs over time.

2.1 Basic Structural Ideas

We suggest three types of storage structure in our approach:

2.1.1 Repository Scheme: this is simply a flat relational scheme that is used to store general details and metadata - such as definitions, constraints, rules, and "dataless" graphs (graphs whose concept nodes map to secondary storage domains where their markers are stored - no markers are stored in the "dataless" graphs, only mappings to secondary domains where the markers can be found) of the knowledge base; it stores data only in certain circumstances. There is only one such instance of a repository, and its structure is binary relational; it is similar to earlier structural approaches (outlined in Bowen and Kocura [1993]). For efficient processing, the repository is read in and converted to a pointer-based representation before anything else is allowed to happen; this allows us to perform operations at main memory speeds, rather than having to access the repository scheme

\(^4\) Although Peirce's "constructivist" approach doesn't readily lend itself to computational implementation.

\(^5\) For reasons of space, this example will be rather briefer than other examples devised by the authors.
continuously - and because the repository only stores general information, the memory required to store it can be quite small in comparison to those KBS which have to be located completely in main memory.

2.1.2 Named Schemes: these are relation schemes that represent graphs, and are schemes to which the user has assigned explicit labels - effectively, they represent the type and relation definitions of conceptual graph theory. Each definition is mapped to a relation scheme of its own, with each concept in the definition being represented by a domain in the scheme. Thus, a definition with twelve concepts is represented by a 12-ary relation scheme. The relationships between these domains are stored by the repository scheme. For example, the type definition for TALL_HANDSOME_PERSON (x):

\[
\text{[PERSON: } *x] - \\
\text{(ATTR)}->[TALLNESS] \\
\text{(ATTR)}->[ATTRACTIVENESS].
\]

would generate the named scheme:

TALL_HANDSOME_PERSON(PERSON,TALLNESS,ATTRACTIVENESS)

and a series of records in the repository scheme which link the domains together and record their location.

Only tuples of named schemes which do not contain null markers are considered to be composite individuals of that scheme, so:

TALL_HANDSOME_PERSON(Tom,*1,*2)

is an instantiation\(^6\), whilst:

TALL_HANDSOME_PERSON(Jerry,*3,NULL)

is not. We can form a definitional hierarchy based on the organisation of labels in the type and relation hierarchies. A good example of this (from Gray [1985]) is shown here:

\(^6\) Elements like *1 and *2 are generic markers that are internally unique - when a database index specifies uniqueness of key, this is a necessary step.
The link between the schemes in the hierarchy is the surrogate P#, which represents the key field and corresponds to the genus marker in Aristotelian definitions - as every TEACHER is still a PERSON, the marker appears there also, and so on.

2.1.3 λ-Schemes: whereas named schemes represent definite types and relations, λ-schemes represent those graphs which aren't named by the user - those assertions of data which comprise the main part of a knowledge-base. For example, if the graph:

\[
\text{[PERSON: Tom]} -
(\text{ATTR})\rightarrow[\text{TALLNESS}]
(\text{LOC})\rightarrow[\text{PLACE: Toms_House}].
\]

doesn't have any corresponding named scheme (which is possible if the graph contains more specialised concepts than existing schemes are equipped to handle, or it contains subgraphs that do not yet have a storage location), a λ-scheme will be generated to store it:

\[
\lambda 1(\text{PERSON,TALLNESS,PLACE})
\]

with the instance (Tom,*4,Toms_House). As with the named schemes, the relationships between the various domains are stored away in the repository scheme. Apart from the fact that the system has to generate a name for such schemes, there would seem to be very little basic difference between these and the named schemes.
We can say that there are four ways in which a λ-scheme can be employed:

(i) "specialisation": those which add extra domains to a previously-known graph but don't restrict the constituent nodes of the graph in any way;
(ii) "restriction": those which add no new domains but restrict one or more of the constituent nodes in a previously-known graph;
(iii) a combination of (i) and (ii).

2.2 Scheme Manipulation

We now outline the operations that we deem necessary for the efficient maintenance and operation of the structures detailed above, and also discuss a number of general issues that are important in efficiency terms.

2.2.1 Named Schemes and Constants: concepts possessing a constant marker in definitions/named schemes are not allocated domains in the scheme, as repeating the constant over and over again would be wasteful of space. Thus, the definition:

WASHINGTON_BUILDING = λx

[BUILDING: *x]->(LOC)->[PLACE: Washington].

would be considered to produce the named scheme:

WASHINGTON_BUILDING(BUILDING)

The information on the constant is retained by the repository scheme and the structures generated from it. This restriction doesn't apply to λ-schemes, as the marker would be considered simply to be an item of data and would be stored somewhere.

2.2.2 Storing graphs within existing schemes: if a graph matches part or all of a graph previously asserted in structural terms, then the data in that graph can be stored in the domains that map to the existing graph and there is no need to spawn a new λ-scheme. The graph may be fragmented into as many parts as are needed to allow the storage, with the proviso that we keep as many concepts grouped together as possible - in order to avoid having to pad out tuples with null values.
There may well be more than one place where the graph can match; for instance, it may match parts of several definitions and a part of a λ-scheme, and a copy of the data is placed in all of them. This does introduce a measure of redundancy, but it is controllable and allows us to make sure that we don't miss some data if we select the wrong occurrence of the graph in the storage schemes.

2.2.3 Spawning of λ-Schemes: if a graph contains information that cannot be placed in existing λ- or named schemes, we need to generate a new λ-scheme to hold this new information. For example, let us say that we only know about the named scheme for TALL_HANDSOME_PERSON, and then introduce the graph:

\[
\text{[PERSON: Christiane]} - \\
(\text{ATTR})-\rightarrow[\text{TALLNESS}] \\
(\text{ATTR})-\rightarrow[\text{ATTRACTIVENESS}] \\
(\text{ATTR})-\rightarrow[\text{WEALTH}].
\]

The subgraph which matches the named scheme is stored there; however, the subgraph:

\[
\text{[PERSON: Christiane]}-\rightarrow(\text{ATTR})-\rightarrow[\text{WEALTH}].
\]

cannot be stored anywhere, and so we have to spawn:

\[
\lambda 1 (\text{PERSON, WEALTH})
\]

in order to store it.

2.2.4 Expansion and Contraction: if a tuple of a named scheme is fully instantiated (ie: it contains no null values), then contraction can occur. All uses of the genus marker in the named scheme are located in the various λ-schemes that exist; as the λ-schemes holding the affected tuples are no longer sufficient to contain all the information of that tuple (precisely because the information in the type field has increased), they are moved to new λ-schemes that can contain the extra information. If no such λ-scheme exists, then one is spawned. For example, say that we have the named scheme TALL_OLD_MAN and a λ-scheme λ1 representing the graph:

\[
\text{[MAN]}-\rightarrow(\text{ATTR})-\rightarrow[\text{TALLNESS}].
\]
Let $\lambda_1$ contain the instantiation:

\[ [\text{MAN: Bill}] \rightarrow (\text{ATTR}) \rightarrow [\text{TALLNESS}] \]

If we now receive information that allows us to instantiate Bill as an OLD_MAN, then the tuple stored in $\lambda_1$ will lose information if it is kept there (because Bill is now an OLD_MAN instead of just a MAN). We therefore move the tuple from $\lambda_1$ to TALL_OLD_MAN, which maintains the most specific conformity of the marker Bill.

2.2.5 $\lambda$-Merging: if two $\lambda$-schemes share a number of common domains and have an exact match between part or all of the two graphs that map to them, then it may be possible to join them into a single $\lambda$-scheme by the operation of $\lambda$-merging. The rationale for such an operation is that it reduces the number of relation schemes that have to be maintained, as well as potentially reducing the number of joins that have to be performed - a highly desirable property$^7$. If a $\lambda$-merge is possible, then it should occur between two $\lambda$-schemes which share the largest number of common domains.

Although the principle of $\lambda$-merging is simple (it resembles a specialised case of the maximal join in conceptual graph theory), the circumstances under which it should occur are not quite so simple - it is pointless to merge $\lambda$-schemes where one can subsume the other. For example, it would make no sense to merge:

\[
\begin{align*}
\lambda_1(\text{MAN, WEALTH}) \\
\lambda_2(\text{OLD_MAN, WEALTH})
\end{align*}
\]

because there would be a loss of information - namely, whether a man was old or not. It is also pointless to join schemes that have very few instances in common, because a merge would result in the formation of many null domains where there was no correspondence between the two schemes.

To counteract this potential waste of space, we suggest a heuristic compromise, whereby two tables are not merged unless they share (for example) 50% of the instances in the joining domains, and where there are more than a specified number of records in each scheme (perhaps a few thousand); this compromise gives us the advantages that we outlined earlier, whilst also cutting back on the number of null value (and thus wasted) cells in the resultant table.

$^7$ As is generally known, the join operation "...is one of the most time-consuming and data-intensive operations in relational query processing" (Mishra and Eich [1992]). It is therefore logical to seek a solution which minimises the joining of relations, and $\lambda$-merging seek to do this.
2.2.6 \(\lambda\)-Splitting: this operation is more or less the opposite of \(\lambda\)-merging, and its job is to split schemes apart that contain too many null values in certain domains. For example, let us say that we have the \(\lambda\)-scheme:

\[
\lambda 1(MAN, TALLNESS, ATTRACTIVENESS, WEALTH)
\]

which corresponds to the graph:

\[
[MAN] -
\quad (ATTR)\rightarrow[TALLNESS]
\quad (ATTR)\rightarrow[ATTRACTIVENESS]
\quad (ATTR)\rightarrow[WEALTH].
\]

If the scheme consists mostly of men who aren't handsome and wealthy (ie: these domains contain lots of null values), but who are merely tall, then we split the scheme into:

\[
\lambda 1(MAN, TALLNESS)
\lambda 2(MAN, ATTRACTIVENESS, WEALTH)
\]

as well as rewriting the mappings in the repository scheme. It is important that the splitting ensures that the resultant schemes can be rejoined to get the original scheme. The heuristic that applies to \(\lambda\)-merging can be applied in reverse - if a group of domains have over 50\% null values, then split off those domains into another scheme. In addition, we can apply another heuristic that states that if the degree of a proposed scheme is less than or equal to two, then the split would result in a binary relational scheme which isn't really worth doing from a join point of view - a set of such schemes would be little better than Datalog (Ullman [1988]) and associated systems, which operate on such a predicate level. Although this approach is rather crude and mechanistic, more statistically-based approaches are being investigated.

2.2.7 \(\lambda\)-Casting: if a new named scheme is introduced that features information present in some existing \(\lambda\)-schemes, we attempt to move the data found in the \(\lambda\)-scheme to the new named scheme. If the \(\lambda\)-scheme exactly matches part or all of the new named scheme, the old \(\lambda\)-scheme is discarded and all of its information transferred to the named scheme\(^8\).

\(^8\) Because it may not be desirable to perform \(\lambda\)-operations whenever it is possible, it might be preferable to execute \(\lambda\)-operations after a set number of user operations (possibly specified by the user).
For example, say that we have the graph:

\[ \text{[MAN]} \rightarrow \text{(ATTR)} \rightarrow \text{[TALLNESS].} \]

and \( \lambda \)-scheme:

\[ \lambda_1(\text{MAN, TALLNESS}) \]

and we then add the named graph:

\[ \text{TALL}_RICH\_\text{MAN} = \lambda x \]

\[ [\text{MAN: } *x] - \]
\[ (\text{ATTR}) \rightarrow [\text{TALLNESS}] \]
\[ (\text{ATTR}) \rightarrow [\text{WEALTH}]. \]

and scheme:

\[ \text{TALL}_RICH\_\text{MAN} (\text{MAN, TALLNESS, WEALTH}) \]

then the domains of the \( \lambda \)-scheme are a subset of the domains of the named scheme and can be transferred to the named domains.

2.2.8 Links between Named Schemes and \( \lambda \)-Schemes: a named scheme stores information from three sources - from explicit declarations of composite individuals, from graph fragmentation over various schemes, and by "implied use". This last source occurs as the result of a named scheme's label being used elsewhere. For example, say that we have the graph:

\[ [\text{MAN: Brian}] \rightarrow (\text{ATTR}) \rightarrow [\text{TALLNESS}]. \]

As well as being stored in various \( \lambda \)-schemes, this graph implicitly implies three things - that a PERSON exists with the identifier "Brian", that an instance of TALLNESS exists, and that an occurrence of ATTR links the two. We can therefore note this in the named schemes:

PERSON(Brian,*5,*6,*7,...)
TALLNESS(*8,*9,*10,...)
ATTR(Brian,*11,*12,...,*8)
All we know about Brian and the others is that they exist - and if they exist, then the other fields in the named schemes must exist also. Therefore, we have to pad out the other domains to say that we know that they must also exist, but that we don't know anything about them beyond that. If we have the opportunity to change any of those generic markers later on, we do so.

2.2.9 Truth and falsity: because the repository scheme maintains the relationship between relation schemes and the graphs that they represent, separate relations can be maintained for negated and non-negated versions of the same graphs. $\lambda$-merging would not apply across a contextual boundary, so graphs that are nested would have only limited scope for $\lambda$-optimisation.

As all rules (which are complex graphs) will be held in the repository, theorem proving mechanisms can select from odd or even schemes, depending on which enclosure is required.

2.2.10 Querying: the method used for performing projection is quite a simple one:

- search the pointer structure (that is produced from the repository scheme) for all projections (and specialisations) of the query graph - as the pointer structure contains all the graphs that are mere "shells" (remember that their data has been stripped out and placed in relational storage), we pay no attention to individual markers at this point;

- for each scheme (either named or $\lambda$) that fits a fragment of the graph, find out which domains map to parts of the graph. If there are markers in a graph fragment, perform a relational select to find the tuple(s) required;

- relationally project out any domains which are irrelevant to the query from the various schemes selected;

- where specialisations have been found of part of a graph, union the various schemes together;

- join the resultant tables together on the appropriate domains, as specified by the maps between the query graph and the underlying schemes.

---

9 This is very different to the use of a null value. The generic marker acknowledges existence, although nothing else is known; the null value denies even the existence.
The query technique follows a pronounced select-project-union-join pattern; this is similar in certain respects to so-called "select-project-join" (SPJ) expressions, whereby the hardest and most expensive algebraic operations are performed last, with earlier operations optimising the later operations as much as possible. Although the introduction of a hierarchical component forces us to break up what could have been single SQL statements (because we have to project and unite tables which are related by specialisation), the general SPJ optimisation principle still holds in the described approach\(^\text{10}\).

### 2.3 Discussion

The approach outlined plays to the strength of the relational model - the efficient manipulation of large sets of regular data. Although it bears a superficial resemblance to the generalisation hierarchy (Sowa [1984]), it is rather more the generation of relation schemes with reference to hierarchies than the classic generalisation hierarchy - the hierarchy is of relation schemes rather than graphs, and each scheme can contain many graph fragments.

The overriding aims of the described approach are to free the user from having to possess knowledge of storage structures, and to continually modify those structures in order to provide cheap queries. To achieve the second aim, it is obvious that we have departed fairly substantially from the classical concept of normalisation in allowing redundancy in the underlying structures. Whereas the basic aim of normalisation is to organise the structures to avoid redundancy, it also increases the fragmentation of those structures; this is in opposition to our aim, and so we have - rather reluctantly, it must be said - put most of the rules of normalisation to one side.

At a certain level, the described approach can be seen as a hierarchical extension of Datalog. More importantly, Datalog (and related schemes which address conceptual graphs, such as Poesio [1987]) works solely at the predicate level - it makes no attempt to minimise the joins required to answer a query, whereas the approach described here actively attempts to minimise joins and works at a higher level than the predicate if possible (due to $\lambda$-merging). The described approach can be seen as universal-relation (in Ullman and Vardi's sense) because:

\(^{10}\) An earlier scheme used the operation of $\lambda$-splitting to maintain "scheme unifiability" - the split would break up schemes in order to maintain identical degree between relation schemes that were related by specialisation. Although this made for a simpler query (which wouldn't have to calculate which domains to strip away before union), it could greatly complicate the manipulations required after an assertion; this is why we have chosen a scheme which is slightly less efficient in query terms, but
• users don't need to know the location of the data in the underlying storage structures.

• the interface between the data structures and the user conforms to the user's view of the world, not to underlying structural concerns.

• abstract queries can be performed. At the most general (and patently too general!) level, the user can use the most general relations and concepts (i.e.: (LINK) and [UNIVERSAL], respectively) if the names aren't known, and projection will be able to locate something. More realistically, a series of answers can be located with incomplete information. Therefore, the requirement to know the exact domain names is reduced.

2.4 An Example of Storage

The aim of this section is give a very brief flavour of the adaptive nature of the approach described above. The graphs and definitions described should be considered as being introduced in the sequence in which they appear below.

a. The definition:

\[
\text{HANDSOME\_MAN} = \lambda x
\]

\[\text{[MAN: *x]} \rightarrow \text{(ATTR)} \rightarrow \text{[ATTRACTIVENESS]}\].

produces these named schemes and repository structures:

\begin{align*}
\text{MAN(UNIV)} \\
\text{ATTRACTIVENESS(UNIV)} \\
\text{HANDSOME\_MAN(MAN,ATTRACTIVENESS)}
\end{align*}

\begin{align*}
\text{UNIV} & : 1, \text{MAN} \\
& : 1, \text{ATTRACTIVENESS} \\
\text{MAN} & : 1, \text{HANDSOME\_MAN} \\
\text{ATTRACTIVENESS} & : 2, \text{HANDSOME\_MAN} \\
\text{ATTR} & : 1 \rightarrow 2, \text{HANDSOME\_MAN}
\end{align*}
b. The graphs:

\[\text{MAN(Tom)}\rightarrow (\text{ATTR})\rightarrow [\text{ATTRACTIVENESS}].\]
\[\text{MAN(Dick)}\rightarrow (\text{ATTR})\rightarrow [\text{ATTRACTIVENESS}].\]

produce these tuples in existing schemes:

\[
\begin{align*}
\text{MAN(Tom)} \\
\text{MAN(Dick)} \\
\text{ATTRACTIVENESS(*1)} \\
\text{ATTRACTIVENESS(*2)} \\
\text{HANDSOME\_MAN(Tom, *1)} \\
\text{HANDSOME\_MAN(Dick, *2)}
\end{align*}
\]

c. The graph:

\[\text{MAN(Harry)} - \]
\[
\begin{align*}
(\text{ATTR})\rightarrow [\text{TALLNESS}] \\
(\text{ATTR})\rightarrow [\text{ATTRACTIVENESS}] \\
(\text{ATTR})\rightarrow [\text{SWARTHINESS}].
\end{align*}
\]

requires the spawning of a \(\lambda\)-scheme to accommodate the information not in \text{HANDSOME\_MAN}:

\[\lambda 1(\text{MAN,TALLNESS,SWARTHINESS})\]

and also spawns these named schemes:

\[
\begin{align*}
\text{TALLNESS(UNIV)} \\
\text{SWARTHINESS(UNIV)}
\end{align*}
\]

It stores these tuples:

\[
\begin{align*}
\text{MAN(Harry)} \\
\text{ATTRACTIVENESS(*3)} \\
\text{TALLNESS(*4)} \\
\text{SWARTHINESS(*5)} \\
\text{HANDSOME\_MAN(Harry, *3)} \\
\lambda 1(Harry,*4,*5)
\end{align*}
\]
and generates these repository structures:

- **UNIV**
  - 1, TALLNESS
  - 1, SWARTHINESS
- **MAN**
  - 1, \( \lambda 1 \)
- **TALLNESS**
  - 2, \( \lambda 1 \)
- **SWARTHINESS**
  - 3, \( \lambda 1 \)
- **ATTR**
  - 1->2, \( \lambda 1 \)
  - 1->3, \( \lambda 1 \)

After the preceding steps, the storage structures and repository structures generated are:

- **MAN**(UNIV)
- **ATTRACTIVENESS**(UNIV)
- **HANDSOME_MAN**(MAN, ATTRACTIVENESS)
- \( \lambda 1 \)(MAN, TALLNESS, SWARTHINESS)
- **TALLNESS**(UNIV)
- **SWARTHINESS**(UNIV)

**UNIV**

- 1, MAN
- 1, ATTRACTIVENESS
- 1, TALLNESS
- 1, SWARTHINESS

**MAN**

- 1, HANDSOME_MAN
- 1, \( \lambda 1 \)

**TALLNESS**

- 2, \( \lambda 1 \)

**SWARTHINESS**

- 3, \( \lambda 1 \)

**ATTRACTIVENESS**

- 2, HANDSOME_MAN

**ATTR**

- 1->2, HANDSOME_MAN
- 1->2, \( \lambda 1 \)
- 1->3, \( \lambda 1 \)

The graph:

\[\text{[PERSON: Larry]} \rightarrow \text{[ATTR]} \rightarrow \text{[WEALTH]}\]

does not exist in any of the structures previously generated; we therefore spawn a new \( \lambda \)-scheme:

\[\text{[PERSON: Larry]} \rightarrow \text{[ATTR]} \rightarrow \text{[WEALTH]}\]
\( \lambda_2(\text{MAN}, \text{WEALTH}) \)

and also this named scheme:

\[ \text{WEALTH}(\text{UNIV}) \]

It stores these tuples:

\[
\begin{align*}
\text{MAN}(\text{Larry}) \\
\text{WEALTH}(\ast 6) \\
\lambda_2(\text{Larry}, \ast 6)
\end{align*}
\]

and generates these repository structures:

\[
\begin{align*}
\text{UNIV} & \rightarrow 1, \text{WEALTH} \\
\text{MAN} & \rightarrow 1, \lambda_2 \\
\text{WEALTH} & \rightarrow 2, \lambda_2 \\
\text{ATTR} & \rightarrow 1 \rightarrow 2, \lambda_2
\end{align*}
\]

So after the preceding steps, the storage structures and repository structures generated are:

\[
\begin{align*}
\text{MAN}(\text{UNIV}) \\
\text{ATTRACTIVENESS}(\text{UNIV}) \\
\text{HANDSOME\_MAN}(\text{MAN}, \text{ATTRACTIVENESS}) \\
\lambda_1(\text{MAN}, \text{TALLNESS}, \text{SWARThINESS}) \\
\text{TALLNESS}(\text{UNIV}) \\
\text{SWARThINESS}(\text{UNIV}) \\
\lambda_2(\text{MAN}, \text{WEALTH}) \\
\text{WEALThI}(\text{UNIV})
\end{align*}
\]

\[
\begin{align*}
\text{UNIV} & \rightarrow 1, \text{MAN} \\
& \rightarrow 1, \text{ATTRACTIVENESS} \\
& \rightarrow 1, \text{TALLNESS} \\
& \rightarrow 1, \text{SWARThINESS} \\
& \rightarrow 1, \text{WEALTH}
\end{align*}
\]
This system is currently being implemented, and may be described (along with algorithmic and test details, and also any modifications to the basic approach) in future papers.

3. References


GRAY, P.M.D. (1985), Logic, Algebra and Databases, Ellis Horwood.


Appendix F (Subappendix A)
Maximal Join Tests

This subappendix contains an edited printout of the routines used to test maximal join in C-GRASS. As all routines are more-or-less the same (except for their size and join sites), we include only a few examples here.

```c
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include "cgclass.h"

static void pr_times (real, tmsstart, tmsend)
    clock_t real;
    struct tms *tmsstart, *tmsend;
{
    long clktck;
    /* Timing Display Routine. */
    clktck = sysconf (_SC_CLK_TCK);
    printf(" real: \$7.7f\n", real / (double) clktck);
    printf(" user: \$7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf(" sys: \$7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf(" child user: \$7.7f\n", (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf(" child sys: \$7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

void MakeFiveDyadUnit (Cl, C2, C3, C4)
    Concept *CI, *C2, *C3, *C4;
{
    /* This routine makes a basic unit of five dyads: */
    /* (R1)->[C2: *b] - */
    /* (R2)<-[C3: *c] - */
    /* (R4)->[C4: *d] - */
    /* (R3)<-{CI: *a}. */
    /* RelationMake does the same for Relation objects as ConceptMake does */
    /* for Concept objects, and AddRNode links a given concept to a given */
    /* relation - it should be noted that two such links are needed to make */
    /* a dyad (one for each link of a concept to the relation). */
    char *StrR1 = "R1\0", *StrR2 = "R2\0", *StrR3 = "R3\0", *StrR4 = "R4\0", *StrR5 = "R5\0";
    R1 = typealloc (Relation);
    R2 = typealloc (Relation);
    R3 = typealloc (Relation);
    R4 = typealloc (Relation);
    R5 = typealloc (Relation);
    RelationMake (StrR1, R1);
    RelationMake (StrR2, R2);
    RelationMake (StrR3, R3);
    RelationMake (StrR4, R4);
    RelationMake (StrR5, R5);
```

RelationMake (StrR4, R4);
RelationMake (StrR5, R5);
AddRNode (Cl, R1, FROM, '\0', 0);
AddRNode (C2, R1, TO, '\0', 0);
AddRNode (C3, R2, FROM, '\0', 0);
AddRNode (C2, R2, TO, '\0', 0);
AddRNode (Cl, R3, FROM, '\0', 0);
AddRNode (C4, R3, TO, '\0', 0);
AddRNode (C3, R4, FROM, '\0', 0);
AddRNode (C4, R4, TO, '\0', 0);
AddRNode (C2, R5, FROM, '\0', 0);
AddRNode (C4, R5, TO, '\0', 0);

void Join5_8010 (TLattice, RLattice, TypeDefs, RelDefs, FnDefs, CanonModel)

LatticeElt *TLattice, *RLattice;
Defn *TypeDefs, *RelDefs, *FnDefs;
ContextListElt *CanonModel;

{ struct tm tmstart, tmsend;
clock_t start, end;
ContextListElt *Complex_u, *Complex_v, *Joined, *ScanJ;
Context *Complex_u_Cxt, *Complex_v_Cxt, *ScanJcxt;
ConceptListElt *Complex_u_SimpleGraphPtr, *Complex_v_SimpleGraphPtr;
Concept *Complex_u, *Complex_v, *Complex_u_SimpleGraphPtr, *Complex_v_SimpleGraphPtr;
char *StrCl = ClO, *StrC2 = C2O, *StrC3 = C3O, *StrC4 = C4O,
char *StrC5 = C5O, *StrC6 = C6O, *StrC7 = C7O, *StrC8 = C8O,
char *StrC9 = C9O, *StrC10 = C10O, *StrC11 = C11O,
char *StrC12 = C12O, *StrC13 = C13O, *StrC14 = C14O,
char *StrC15 = C15O, *StrC16 = C16O, *StrC17 = C17O,
char *StrC18 = C18O, *StrC19 = C19O, *StrC20 = C20O,
char *StrC21 = C21O, *StrC22 = C22O, *StrC23 = C23O,
char *StrC24 = C24O, *StrC25 = C25O, *StrC26 = C26O,
char *StrC27 = C27O, *StrC28 = C28O, *StrC29 = C29O,
char *StrC30 = C30O, *StrC31 = C31O, *StrC32 = C32O,
char *StrC33 = C33O, *StrC34 = C34O, *StrC35 = C35O,
char *StrC36 = C36O, *StrC37 = C37O, *StrC38 = C38O,
char *StrC39 = C39O, *StrC40 = C40O,
char *StrC41 = C41O, *StrC42 = C42O, *StrC43 = C43O,
char *StrC44 = C44O, *StrC45 = C45O, *StrC46 = C46O,
char *StrC47 = C47O, *StrC48 = C48O, *StrC49 = C49O,
char *StrC50 = C50O, *StrC51 = C51O, *StrC52 = C52O,
char *StrC53 = C53O, *StrC54 = C54O, *StrC55 = C55O,
char *StrC56 = C56O, *StrC57 = C57O, *StrC58 = C58O,
char *StrC59 = C59O, *StrC60 = C60O, *StrC61 = C61O,
char *StrC62 = C62O, *StrC63 = C63O, *StrC64 = C64O,
char *StrC65 = C65O, *StrC66 = C66O, *StrC67 = C67O,
char *StrC68 = C68O, *StrC69 = C69O, *StrC70 = C70O,
char *StrC71 = C71O, *StrC72 = C72O, *StrC73 = C73O,
char *StrC74 = C74O, *StrC75 = C75O, *StrC76 = C76O,
char *StrC77 = C77O, *StrC78 = C78O, *StrC79 = C79O,
char *StrC80 = C80O, *StrC81 = C81O, *StrC82 = C82O,
char *StrC83 = C83O, *StrC84 = C84O, *StrC85 = C85O,
char *StrC86 = C86O, *StrC87 = C87O, *StrC88 = C88O,
char *StrC89 = C89O, *StrC90 = C90O, *StrC91 = C91O,
char *StrC92 = C92O, *StrC93 = C93O, *StrC94 = C94O,
char *StrC95 = C95O, *StrC96 = C96O, *StrC97 = C97O,
char *StrC98 = C98O, *StrC99 = C99O, *StrC100 = C100O;
*/
/* This is Test Program J5_8010 - joining two graphs (u and v - one eighty dyads in size.*/
/* the other ten dyads on a five-dyad join site. This program sets up the two graphs u */
/* and v, then submits them to the Maximal Join operation and displays the result. */
/* The first step is to make graph u... out of sixteen basic units of four concepts and */
/* five relations apiece... */
/* *****************************************************************************/
C1u = typealloc (Concept);
C2u = typealloc (Concept);
C3u = typealloc (Concept);
C4u = typealloc (Concept);
C5u = typealloc (Concept);
C6u = typealloc (Concept);
C7u = typealloc (Concept);
C8u = typealloc (Concept);
C9u = typealloc (Concept);
C10u = typealloc (Concept);
C11u = typealloc (Concept);
C12u = typealloc (Concept);
C13u = typealloc (Concept);
C14u = typealloc (Concept);
C15u = typealloc (Concept);
C16u = typealloc (Concept);
C17u = typealloc (Concept);
C18u = typealloc (Concept);
C19u = typealloc (Concept);
C20u = typealloc (Concept);
C21u = typealloc (Concept);
C22u = typealloc (Concept);
C23u = typealloc (Concept);
C24u = typealloc (Concept);
C25u = typealloc (Concept);
C26u = typealloc (Concept);
C27u = typealloc (Concept);
C28u = typealloc (Concept);
C29u = typealloc (Concept);
C30u = typealloc (Concept);
C31u = typealloc (Concept);
C32u = typealloc (Concept);
C33u = typealloc (Concept);
C34u = typealloc (Concept);
C35u = typealloc (Concept);
C36u = typealloc (Concept);
C37u = typealloc (Concept);
C38u = typealloc (Concept);
C39u = typealloc (Concept);
C40u = typealloc (Concept);
ConceptMake (StrCl, Reflu, Clu, FALSE, '"D', '"0');
ConceptMake (StrCl, Ref2u, C2u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref3u, C3u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref4u, C4u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref5u, C5u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref6u, C6u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref7u, C7u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref8u, C8u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref9u, C9u, FALSE, '"0', '"0');
ConceptMake (StrCl, Ref10u, ClDu, FALSE, '"0', '"0');
MakeFiveDyadUnit (elu, C2u, C3u, C4u);
MakeFiveDyadUnit (C5u, C6u, C7u, C4u);
MakeFiveDyadUnit (C9u, C8u, C3u, C10u);
MakeFiveDyadUnit (C12u, C11u, C7u, C10u);
MakeFiveDyadUnit (C13u, C6u, C15u, C14u);
MakeFiveDyadUnit (C17u, C11u, C15u, C16u);
MakeFiveDyadUnit (C18u, C19u, C20u, C14u);
MakeFiveDyadUnit (C22u, C21u, C20u, C16u);
MakeFiveDyadUnit (C24u, C23u, C26u, C27u);
MakeFiveDyadUnit (C29u, C30u, C27u);
MakeFiveDyadUnit (C33u, C35u, C27u);
MakeFiveDyadUnit (C36u, C35u, C37u);
MakeFiveDyadUnit (C39u, C36u, C37u);

/* Make up graph v here - two five-units... */
C1v = typealloc (Concept);
C2v = typealloc (Concept);
C3v = typealloc (Concept);
C4v = typealloc (Concept);
C5v = typealloc (Concept);
C6v = typealloc (Concept);
C7v = typealloc (Concept);

ConceptMake (StrC1, Ref1v, C1v, FALSE, '0', '0');
ConceptMake (StrC2, Ref2v, C2v, FALSE, '0', '0');
ConceptMake (StrC3, Ref3v, C3v, FALSE, '0', '0');
ConceptMake (StrC4, Ref4v, C4v, FALSE, '0', '0');
ConceptMake (StrC5, Ref5v, C5v, FALSE, '0', '0');
ConceptMake (StrC6, Ref6v, C6v, FALSE, '0', '0');
ConceptMake (StrC7, Ref7v, C7v, FALSE, '0', '0');

C1v->Next = C2v;
C2v->Next = C3v;
C3v->Next = C4v;
C4v->Next = C5v;
C5v->Next = C6v;
C6v->Next = C7v;
C7v->Next = 0;
MakeFiveDyadUnit (C1v, C2v, C3v, C4v);
MakeFiveDyadUnit (C5v, C6v, C7v, C4v);

/********************************-*************-****************************-******/
/* Display the joined graph and exit the program before anything else can happen. */
**************************************************.*.*** •• *.* •••• *.*** •• *-*.* ••• *** ••
printf (*
TEST: JoinS_BOlD...

*);
Graph_Display_Gateway (GRAPH, '0', TypeDefs, RelDefs, FnDefs, Joined, CanonModel, TLattice, RLattice);

start = times (&tmsstart);
Joined = MaximalJoin (Complex_u, Complex_v, TLattice, RLattice, 0, 0, FALSE, FALSE, TRUE);
end = times (&tmsend);

for (ScanJ = Joined; ScanJ != 0; ScanJ = ScanJ->NextInList){
  ScanJCxt = ScanJ->PointedAt;
  for (ScanJM = ScanJCxt->ListOfMembers; ScanJM != 0; ScanJM = ScanJM->NextInList){
    Backtrack_Over_Flags (ScanJM->PointedAt);
  }
}

printf (*\nTEST: Join5_6010...

*);
Graph_Display_Gateway (GRAPH, '0', TypeDefs, RelDefs, FnDefs, Joined, CanonModel, TLattice, RLattice);
void Join5_4010 (TLattice, RLattice,TypeDefs, RelDefs, FnDefs, CanonModel)

LatticeElT *Lattice, *RLattice;
Defn *TypeDefs, *RelDefs, *FnDefs;
ContextListElT *CanonModel;

{ struct tms tmsstart, tmsend;
  clock_t start, end;
  ContextListElT *Complex_u, *Complex_v, *Joined, *ScanJ;
  Context *Complex_u_Cxt, *Complex_v_Cxt, *ScanJ_H;
  ConceptListElT *Complex_u_SimpleGraphPtr, *Complex_v_SimpleGraphPtr;
  char *StrCl = '-', *StrC2 = '-', *StrC3 = '-', *StrC4 = '-', *StrC5 = '-', *StrC6 = '-', *StrC7 = '-', *StrC8 = '-', *StrC9 = '-', *StrClu = '-', *StrC2u = '-', *StrC3u = '-', *StrC4u = '-', *StrC5u = '-', *StrC6u = '-', *StrC7u = '-', *StrC8u = '-', *StrC9u = '-', *StrC10u = '-', *StrCl1u = '-', *StrC12u = '-', *StrC13u = '-', *StrC14u = '-', *StrC15u = '-', *StrC16u = '-', *StrC17u = '-', *StrC18u = '-', *StrC19u = '-', *StrC20u = '-', *StrC21u = '-', *StrC22u = '-';
  char *RefIu = '-', *Ref2u = '-', *Ref3u = '-', *Ref4u = '-', *Ref5u = '-', *Ref6u = '-', *Ref7u = '-', *Ref8u = '-', *Ref9u = '-', *Ref10u = '-', *Ref11u = '-', *Ref12u = '-', *Ref13u = '-', *Ref14u = '-', *Ref15u = '-', *Ref16u = '-', *Ref17u = '-', *Ref18u = '-', *Ref19u = '-', *Ref20u = '-', *Ref21u = '-', *Ref22u = '-';
  char *RefIv = '-', *Ref2v = '-', *Ref3v = '-', *Ref4v = '-', *Ref5v = '-', *Ref6v = '-', *Ref7v = '-', *Ref8v = '-', *Ref9v = '-', *Ref10v = '-', *Ref11v = '-', *Ref12v = '-', *Ref13v = '-', *Ref14v = '-', *Ref15v = '-', *Ref16v = '-', *Ref17v = '-', *Ref18v = '-', *Ref19v = '-', *Ref20v = '-', *Ref21v = '-', *Ref22v = '-';

  printf ("\nTime taken (seconds) to join u and v:\n\n");
  pr_times (end-start, &tmsstart, &tmsend);
}

Clu typealloc (Concept);
C2u typealloc (Concept);
C3u typealloc (Concept);
C4u typealloc (Concept);
CSu typealloc (Concept);
C6u typealloc (Concept);
C7u typealloc (Concept);
C8u typealloc (Concept);
C9u typealloc (Concept);
CIOu typealloc (Concept);
Cllu typealloc (Concept);
C12u typealloc (Concept);
C13u typealloc (Concept);
C14u = typealloc (Concept);
C15u = typealloc (Concept);
C16u typealloc (Concept);
C17u typealloc (Concept);
C18u typealloc (Concept);
C19u = typealloc (Concept);
C20u = typealloc (Concept);
C21u typealloc (Concept);
C22u = typealloc (Concept);

ConceptMake (StrCl, RefIu, Clu, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref2u, C2u, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref3u, C3u, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref4u, C4u, FALSE, '\0', '\0');
ConceptMake (StrC5, Ref5u, C5u, FALSE, '\0', '\0');
ConceptMake (StrC6, Ref6u, C6u, FALSE, '\0', '\0');
ConceptMake (StrC7, Ref7u, C7u, FALSE, '\0', '\0');
ConceptMake (StrC8, Ref8u, C8u, FALSE, '\0', '\0');
ConceptMake (StrC9, Ref9u, C9u, FALSE, '\0', '\0');
ConceptMake (StrCI, Ref10u, C10u, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref11u, C11u, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref12u, C12u, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref13u, C13u, FALSE, '\0', '\0');
ConceptMake (StrC5, Ref14u, C14u, FALSE, '\0', '\0');
ConceptMake (StrC6, Ref15u, C15u, FALSE, '\0', '\0');
ConceptMake (StrC7, Ref16u, C16u, FALSE, '\0', '\0');
ConceptMake (StrC8, Ref17u, C17u, FALSE, '\0', '\0');
ConceptMake (StrC9, Ref18u, C18u, FALSE, '\0', '\0');
ConceptMake (StrC1, Ref19u, C19u, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref20u, C20u, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref21u, C21u, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref22u, C22u, FALSE, '\0', '\0');

C1u->Next = C2u;
C2u->Next = C3u;
C3u->Next = C4u;
C4u->Next = C5u;
C5u->Next = C6u;
C6u->Next = C7u;
C7u->Next = C8u;
C8u->Next = C9u;
C9u->Next = C10u;
C10u->Next = C11u;
C11u->Next = C12u;
C12u->Next = C13u;
C13u->Next = C14u;
C14u->Next = C15u;
C15u->Next = C16u;
C16u->Next = C17u;
C17u->Next = C18u;
C18u->Next = C19u;
C19u->Next = C20u;
C20u->Next = C21u;
C21u->Next = C22u;
MakeFiveDyadUnit (C1u, C2u, C3u, C4u);
MakeFiveDyadUnit (C5u, C6u, C7u, C4u);
MakeFiveDyadUnit (C9u, C8u, C3u, C10u);
MakeFiveDyadUnit (C12u, C11u, C7u, C10u);
MakeFiveDyadUnit (C13u, C12u, C15u, C14u);
MakeFiveDyadUnit (C17u, C16u, C20u, C14u);
MakeFiveDyadUnit (C18u, C19u, C20u, C14u);
MakeFiveDyadUnit (C22u, C21u, C20u, C16u);

/* Make up graph v here - two five-units... */

C1v = typealloc (Concept);
C2v = typealloc (Concept);
C3v = typealloc (Concept);
C4v = typealloc (Concept);
C5v = typealloc (Concept);
C6v = typealloc (Concept);
C7v = typealloc (Concept);
ConceptMake (StrC1, Ref1v, C1v, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref2v, C2v, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref3v, C3v, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref4v, C4v, FALSE, '\0', '\0');
ConceptMake (StrC5, Ref5v, C5v, FALSE, '\0', '\0');
ConceptMake (StrC6, Ref6v, C6v, FALSE, '\0', '\0');
ConceptMake (StrC7, Ref7v, C7v, FALSE, '\0', '\0');
C1v->Next = C2v;
C2v->Next = C3v;
C3v->Next = C4v;
C4v->Next = C5v;
C5v->Next = C6v;
C6v->Next = C7v;

C1v->Next = C2v;
C2v->Next = C3v;
C3v->Next = C4v;
C4v->Next = C5v;
C5v->Next = C6v;
C6v->Next = C7v;
C7v->Next = 0;
MakeFiveDyadUnit (C1v, C2v, C3v, C4v);
MakeFiveDyadUnit (C5v, C6v, C7v, C4v);

Complex_u = typealloc (ContextListEl);t
Complex_v = typealloc (ContextListEl);
Complex_u_Cxt = typealloc (Context);
Complex_u_SimpleGraphPtr = typealloc (ContextListEl);
Complex_u->PointedAt = Complex_u_Cxt;
Complex_u->NextInList = 0;
Complex_u_Cxt->Depth = 0;
Complex_u_Cxt->ListOfMembers = Complex_u_SimpleGraphPtr;
Complex_u_SimpleGraphPtr->PointedAt = C1u;
Complex_u_SimpleGraphPtr->NextInList = 0;
printf ('Graph u ...

M);
Graph_Display_Gateway (GRAPH, '\0', TypeDefs, RelDefs, FnDefs, Complex_u, CanonModel, TLattice, RLattice);

Complex_v = typealloc (ContextListEl);
Complex_v_Cxt = typealloc (Context);
Complex_v_SimpleGraphPtr = typealloc (ContextListEl);
Complex_v->PointedAt = Complex_v_Cxt;
Complex_v->NextInList = 0;
Complex_v_Cxt->Depth = 0;
Complex_v_Cxt->ListOfMembers = Complex_v_SimpleGraphPtr;
Complex_v_SimpleGraphPtr->PointedAt = C1v;
Complex_v_SimpleGraphPtr->NextInList = 0;
printf ('Graph v ...

M);
Graph_Display_Gateway (GRAPH, '\0', TypeDefs, RelDefs, FnDefs, Complex_v, CanonModel, TLattice, RLattice);

start = times (etimesstart);
Joined = MaximalJoin (Complex_u, Complex_v, TLattice, RLattice, 0, 0, FALSE, FALSE, TRUE);
end = times (etimesend);

for (ScanJ = Joined; ScanJ != 0; ScanJ = ScanJ->NextInList)
  for (ScanJM = ScanJ->PointedAt)
    for (ScanJN = ScanJM->ListOfMembers; ScanJN != 0; ScanJN = ScanJN->NextInList)
      Backtrack_Over_Flags (ScanJN->PointedAt);
void Join5_2010 (TLattice, RLattice, TypeDefs, RelDefs, FnDefs, CanonModel)
{
    LatticeElt *TLattice, *RLattice;
    Defn *TypeDefs, *RelDefs, *FnDefs;
    ContextListElt *CanonModel;
    struct tms tmsstart, tmsend;
    clock_t start, end;
    ContextListElt *Complex_u, *Complex_v, *Joined, *ScanJ,
    *Complex_u_Cxt, *Complex_v_Cxt, *ScanJCxt,
    ConceptListElt *Complex_u_SimpleGraphPtr, *Complex_v_SimpleGraphPtr,
    *ScanJMCnt;
    ConceptMake (StrCl, Reflu, C1u, FALSE, '\0', '\0');
    ConceptMake (StrC2, Ref2u, C2u, FALSE, '\0', '\0');
    ConceptMake (StrC3, Ref3u, C3u, FALSE, '\0', '\0');
    ConceptMake (StrC4, Ref4u, C4u, FALSE, '\0', '\0');
    ConceptMake (StrC5, Ref5u, C5u, FALSE, '\0', '\0');
    ConceptMake (StrC6, Ref6u, C6u, FALSE, '\0', '\0');
    ConceptMake (StrC7, Ref7u, C7u, FALSE, '\0', '\0');
    ConceptMake (StrC8, Ref8u, C8u, FALSE, '\0', '\0');
    ConceptMake (StrC9, Ref9u, C9u, FALSE, '\0', '\0');
    ConceptMake (StrC10, Ref10u, C10u, FALSE, '\0', '\0');
    ConceptMake (StrC11, Ref11u, C11u, FALSE, '\0', '\0');
    ConceptMake (StrC12, Ref12u, C12u, FALSE, '\0', '\0');
    C1u->Next = C2u;
    C2u->Next = C3u;
    C3u->Next = C4u;
    C4u->Next = C5u;
    C5u->Next = C6u;
    C6u->Next = C7u;
    C7u->Next = C8u;
    C8u->Next = C9u;
    C9u->Next = C10u;
    C10u->Next = C11u;
    C11u->Next = C12u;
    MakeFiveDyadUnit (C1u, C2u, C3u, C4u);
    MakeFiveDyadUnit (C5u, C6u, C7u, C8u);
    MakeFiveDyadUnit (C9u, C8u, C3u, C10u);
    MakeFiveDyadUnit (C12u, C11u, C7u, C10u);
    /* Make up graph v here - two five-units... */
    Clu = typealloc (Concept);
    C2v = typealloc (Concept);
C3\text{v} = \text{typealloc}\left( \text{Concept} \right);
C4\text{v} = \text{typealloc}\left( \text{Concept} \right);
C5\text{v} = \text{typealloc}\left( \text{Concept} \right);
C6\text{v} = \text{typealloc}\left( \text{Concept} \right);
C7\text{v} = \text{typealloc}\left( \text{Concept} \right);

\text{ConceptMake}\left( \text{StrCl}, \text{Ref1v, C1v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC2}, \text{Ref2v, C2v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC3}, \text{Ref3v, C3v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC4}, \text{Ref4v, C4v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC5}, \text{Ref5v, C5v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC6}, \text{Ref6v, C6v, FALSE, '0', '0'} \right);
\text{ConceptMake}\left( \text{StrC7}, \text{Ref7v, C7v, FALSE, '0', '0'} \right);

\text{Clv}\rightarrow\text{Next C2v} ;
\text{C2v}\rightarrow\text{Next C3v} ;
\text{C3v}\rightarrow\text{Next C4v} ;
\text{C4v}\rightarrow\text{Next C5v} ;
\text{C5v}\rightarrow\text{Next } = \text{C6v} ;
\text{C6v}\rightarrow\text{Next C7v} ;
\text{C7v}\rightarrow\text{Next } = 0 ;
\text{MakeFiveDyadUnit}\left( \text{Clv, C2v, C3v, C4v} \right) ;
\text{MakeFiveDyadUnit}\left( \text{C5v, C6v, C7v, C4v} \right) ;

/*******/
/*******/

\text{Complex_u} = \text{typealloc}\left( \text{ContextListElt} \right);
\text{Complex_u_Cxt} = \text{typealloc}\left( \text{Context} \right);
\text{Complex_u_SimpleGraphPtr} = \text{typealloc}\left( \text{ConceptListElt} \right);
\text{Complex_u}\rightarrow\text{pointedAt} = \text{Complex_u_Cxt} ;
\text{Complex_u}\rightarrow\text{NextInList} = 0 ;
\text{Complex_u_Cxt}\rightarrow\text{Depth} = 0 ;
\text{Complex_u_Cxt}\rightarrow\text{ListOfMembers} = \text{Complex_u_SimpleGraphPtr} ;
\text{Complex_u_SimpleGraphPtr}\rightarrow\text{pointedAt} = \text{Clu} ;
\text{Complex_u_SimpleGraphPtr}\rightarrow\text{NextInList} = 0 ;

\text{Graph_Display_Gateway}\left( \text{GRAPH, '0', TypeDefs, RelDefs, FnDefs, Complex_u, CanonModel, TLattice, RLattice} \right) ;

\text{Complex_v} = \text{typealloc}\left( \text{ContextListElt} \right);
\text{Complex_v_Cxt} = \text{typealloc}\left( \text{Context} \right);
\text{Complex_v_SimpleGraphPtr} = \text{typealloc}\left( \text{ConceptListElt} \right);
\text{Complex_v}\rightarrow\text{pointedAt} = \text{Complex_v_Cxt} ;
\text{Complex_v}\rightarrow\text{NextInList} = 0 ;
\text{Complex_v_Cxt}\rightarrow\text{Depth} = 0 ;
\text{Complex_v_Cxt}\rightarrow\text{ListOfMembers} = \text{Complex_v_SimpleGraphPtr} ;
\text{Complex_v_SimpleGraphPtr}\rightarrow\text{pointedAt} = \text{Clv} ;
\text{Complex_v_SimpleGraphPtr}\rightarrow\text{NextInList} = 0 ;

\text{Graph_Display_Gateway}\left( \text{GRAPH, '0', TypeDefs, RelDefs, FnDefs, Complex_v, CanonModel, TLattice, RLattice} \right) ;

\text{start} = \text{times}\left( \text{&tmsstart} \right) ;
\text{Joined} = \text{MaximalJoin}\left( \text{Complex_u, Complex_v, TLattice, RLattice, 0, 0, FALSE, FALSE, TRUE} \right) ;
\text{end} = \text{times}\left( \text{&tmsend} \right) ;

\text{for} \ \left( \text{ScanJ} = \text{Joined, ScanJ} = 0 ; \text{ScanJ} = \text{ScanJ}\rightarrow\text{NextInList} \right) \{
\text{ScanJ}\rightarrow\text{pointedAt} = \text{Clu} ;
\text{for} \ \left( \text{ScanJM} = \text{ScanJ}\rightarrow\text{ListOfMembers, ScanJM} = 0 ; \text{ScanJM} = \text{ScanJM}\rightarrow\text{NextInList} \right) \{
\text{Backtrack_Over_Flags}\left( \text{ScanJM}\rightarrow\text{pointedAt} \right) ;
\text{ScanJM}\rightarrow\text{NextInList} ;
\} 
\}

/*************/

/* Display the joined graph and exit the program before anything else can happen. */

/test: JoinS_2010... Joined Graph...

*/

print\left( 'Time taken (seconds) to join u and v:

-'ight) ;
\text{pr_times}\left( \text{end} - \text{start}, \text{&tmsstart}, \text{&tmsend} \right) ;

\text{void MakeTestLattices}\left( \text{TL, RL} \right) ;

\text{LatticeEl}t\ast \text{TL}, \ast \text{RL} ;

\left\{ \begin{array}{l}
\text{LatticeEl}t \ast \text{MetaS, MetaI}, \ast \text{MetaN, C1, C2, C3, C4, R1, R2, R3, R4, R5;}
\text{LatticeEl}t\ast \text{MetaSPtr, MetaIPtr, MetaNPtr;}
\text{LatticeEl}t\ast \text{C1Ptr, C2Ptr, C3Ptr, C4Ptr, R1Ptr, R2Ptr, R3Ptr, R4Ptr, R5Ptr;}
\text{char} \ast \text{StrCl} = \text{C1\"}, \ast \text{StrC2} = \text{C2\"}, \ast \text{StrC3} = \text{C3\"}, \ast \text{StrC4} = \text{C4\"};
\text{char} \ast \text{StrR1} = \text{R1\"}, \ast \text{StrR2} = \text{R2\"}, \ast \text{StrR3} = \text{R3\"}, \ast \text{StrR4} = \text{R4\"}, \ast \text{StrR5} = \text{R5\"};
\text{char} \ast \text{String} = \text{STRING\"}, \ast \text{Indv} = \text{INDIVIDUAL\"}, \ast \text{Numb} = \text{NUMERIC\"};
\end{array} \right. 

/*************/
/* Simply makes the toy lattices: */
for use in tests.

MetaS = typealloc (LatticeElt);
strncpy (MetaS->Label, '\0', 25);
strncpy (MetaS->Label, String, 7);
MetaS->SubLabels = 0;
MetaI = typealloc (LatticeElt);
strncpy (MetaI->Label, '\0', 25);
strncpy (MetaI->Label, Indv, 12);
MetaI->SubLabels = 0;
MetaN = typealloc (LatticeElt);
strncpy (MetaN->Label, '\0', 25);
strncpy (MetaN->Label, Numb, 8);
MetaN->SubLabels = 0;
MetaSPtr = typealloc (LatticeEltPtr);
MetaIPtr = typealloc (LatticeEltPtr);
MetaNPtr = typealloc (LatticeEltPtr);
MetaSPtr->PointedAt = MetaS;
MetaSPtr->ArcType = ARISTO;
MetaSPtr->NextInList = MetaIPtr;
MetaIPtr->PointedAt = MetaI;
MetaIPtr->ArcType = ARISTO;
MetaIPtr->NextInList = MetaNPtr;
MetaNPtr->PointedAt = MetaN;
MetaNPtr->ArcType = ARISTO;
MetaNPtr->NextInList = 0;
TL->SubLabels = MetaSPtr;

C1 = typealloc (LatticeElt);
strncpy (C1->Label, '\0', 25);
strncpy (C1->Label, Strc1, 3);
C2 = typealloc (LatticeElt);
strncpy (C2->Label, '\0', 25);
strncpy (C2->Label, Strc2, 3);
C3 = typealloc (LatticeElt);
strncpy (C3->Label, '\0', 25);
strncpy (C3->Label, Strc3, 3);
C4 = typealloc (LatticeElt);
strncpy (C4->Label, '\0', 25);
strncpy (C4->Label, Strc4, 3);
C1Ptr = typealloc (LatticeEltPtr);
C1Ptr->PointedAt = C1;
C1Ptr->ArcType = ARISTO;
C2Ptr = typealloc (LatticeEltPtr);
C2Ptr->PointedAt = C2;
C2Ptr->ArcType = ARISTO;
C3Ptr = typealloc (LatticeEltPtr);
C3Ptr->PointedAt = C3;
C3Ptr->ArcType = ARISTO;
C4Ptr = typealloc (LatticeEltPtr);
C4Ptr->PointedAt = C4;
C4Ptr->ArcType = ARISTO;
MetaI->SubLabels = C1Ptr;
C1Ptr->NextInList = C2Ptr;
C2Ptr->NextInList = C3Ptr;
C3Ptr->NextInList = C4Ptr;
C4Ptr->NextInList = 0;

R1 = typealloc (LatticeElt);
strncpy (R1->Label, '\0', 25);
strncpy (R1->Label, StrR1, 3);
R2 = typealloc (LatticeElt);
strncpy (R2->Label, '\0', 25);
strncpy (R2->Label, StrR2, 3);
R3 = typealloc (LatticeElt);
strncpy (R3->Label, '\0', 25);
strncpy (R3->Label, StrR3, 3);
R4 = typealloc (LatticeElt);
strncpy (R4->Label, '\0', 25);
strncpy (R4->Label, StrR4, 3);
R5 = typealloc (LatticeElt);
strncpy (R5->Label, '\0', 25);
strncpy (R5->Label, StrR5, 3);
R1Ptr = typealloc (LatticeEltPtr);
R1Ptr->PointedAt = R1;
R1Ptr->ArcType = LOGICAL;
R2Ptr = typealloc (LatticeEltPtr);
R2Ptr->PointedAt = R2;
R2Ptr->ArcType = LOGICAL;
R3Ptr = typealloc (LatticeEltPtr);
R3Ptr->PointedAt = R3;
R3Ptr->ArcType = LOGICAL;
R4Ptr = typealloc (LatticeEltPtr);
R4Ptr->PointedAt = R4;
R4Ptr->ArcType = LOGICAL;
R5Ptr = typealloc (LatticeEltPtr);
R5Ptr->PointedAt = R5;
R5Ptr->ArcType = LOGICAL;
R1->SubLabels = R1Ptr;
R2Ptr->NextInList = R2Ptr;
R2Ptr->NextInList = R3Ptr;
R3Ptr->NextInList = R4Ptr;
R4Ptr->NextInList = R5Ptr;
R5Ptr->NextInList = 0;

void main ()
{
    Defn *TDL, *RDL, *FDL;
    ContextListElt *SQG;
    LatticeElt *TLattice, *RLattice;
    char *Nul = "\0", *StrU = "UNIVERSAL\0";
    TDL = typealloc (Defn);
    RDL = typealloc (Defn);
    FDL = typealloc (Defn);
    strncpy (TDL->DefName, Nul, 25);
    TDL->NextInList = 0;
    strncpy (RDL->DefName, Nul, 25);
    RDL->NextInList = 0;
    strncpy (FDL->DefName, Nul, 25);
    FDL->NextInList = 0;
    SQG = typealloc (ContextListElt);
    SQG->PointedAt = 0;
    SQG->NextInList = 0;
    TLattice = typealloc (LatticeElt);
    RLattice = typealloc (LatticeElt);
    strncpy (TLattice->Label, Nul, 25);
    strncpy (TLattice->Label, StrU, 9);
    TLattice->SubLabels = 0;
    strncpy (RLattice->Label, Nul, 25);
    strncpy (RLattice->Label, StrU, 9);
    RLattice->SubLabels = 0;
    MakeTestLattices (TLattice, RLattice);
    JoinS_1010 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1020 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1030 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1040 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1050 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1060 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1070 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1080 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1090 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1010 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1020 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1030 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1040 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1050 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1060 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1070 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1080 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    JoinS_1090 (TLattice, RLattice, TDL, RDL, FDL, SQG);
    exit(0);
}
Appendix F (Subappendix B)

Copy Tests

This subappendix contains an edited printout of the routines used to test graph copying in C-GRASS. As all routines are more-or-less the same (except for their size), we include only a couple of examples here.

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include "defines.h"
#include "cgclass.h"

static void pr_times (real, tmsstart, tmsend)
    clock_t real;
    struct tms *tmsstart, *tmsend;
{
    static long clktck = 0;

    /*******************************************************************************/
    /* Timing Display Routine. */
    /*******************************************************************************/
    clktck = sysconf (_SC_CLK_TCK);
    printf("real: %7.7f\n", (double) clktck);
    printf("user: %7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("sys: %7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("child: %7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

void MakeFiveDyadUnit (Cl, C2, C3, C4, Cl, C2, C3, C4
    Concept *Cl, *C2, *C3, *C4;

    /******************************************************************************/
    /* This routine makes a basic unit of five dyads: */
    /******************************************************************************/
    RI typealloc (Relation);
    R2 typealloc (Relation);
    R3 typealloc (Relation);
    R4 = typealloc (Relation);
    RS = typealloc (Relation);
    RelationMake (StrRI, RI);
    RelationMake (StrR2, R2);
    RelationMake (StrR3, R3);
    RelationMake (StrR4, R4);
    RelationMake (StrR5, RS);
```
void CopylO (TLattice, RLattice, TypeDefs, RelDefs, FnDefs, CanonModel)

LatticeElt *TLattice, *RLattice;
Defn *TypeDefs, *RelDefs, *FnDefs;
ContextListElt *CanonModel;
{
    struct tms tmsstart, tmsendi;
    clock_t start, end;
    ContextListEl..*NewG, *ToBeCopied;
    Context *ToBeCopiedCxt;
    ConceptListElt *ToBeCopiedSimpleGraphPtr;
    C1 = typealloc (Concept);
    C2 = typealloc (Concept);
    C3 = typealloc (Concept);
    C4 = typealloc (Concept);
    C5 = typealloc (Concept);
    C6 = typealloc (Concept);
    C7 = typealloc (Concept);
    ConceptMake (StrCl, Ref1, C1, FALSE, '\0', '\0');
    ConceptMake (StrC2, Ref2, C2, FALSE, '\0', '\0');
    ConceptMake (StrC3, Ref3, C3, FALSE, '\0', '\0');
    ConceptMake (StrC4, Ref4, C4, FALSE, '\0', '\0');
    ConceptMake (StrC5, Ref5, C5, FALSE, '\0', '\0');
    ConceptMake (StrC6, Ref6, C6, FALSE, '\0', '\0');
    ConceptMake (StrC7, Ref7, C7, FALSE, '\0', '\0');

    ToBeCopied = typealloc (ContextListElt);
    ToBeCopiedCxt = typealloc (Context);
    ToBeCopiedSimpleGraphPtr = typealloc (ConceptListElt);

    ToBeCopied->NextlnList = ToBeCopiedCxt;
    ToBeCopied->NextlnList = 0;
    ToBeCopiedCxt->NextlnList = ToBeCopiedSimpleGraphPtr;
    ToBeCopiedSimpleGraphPtr->NextlnList = C1;
    ToBeCopiedSimpleGraphPtr->NextlnList = 0;

    start = times (&tmsstart);
    NewG = ShadowCopy (ToBeCopied, 0);
    end = times (&tmsendi);

    printf("Testing speed of copying of a graph composed from 10 dyads. It sets up two basic 5-dyad graph cells and then copies them.
    
    C1 = typealloc (Concept);
    C2 = typealloc (Concept);
    C3 = typealloc (Concept);
    C4 = typealloc (Concept);
    C5 = typealloc (Concept);
    C6 = typealloc (Concept);
    C7 = typealloc (Concept);
    ConceptMake (StrCl, Ref1, C1, FALSE, '\0', '\0');
    ConceptMake (StrC2, Ref2, C2, FALSE, '\0', '\0');
    ConceptMake (StrC3, Ref3, C3, FALSE, '\0', '\0');
    ConceptMake (StrC4, Ref4, C4, FALSE, '\0', '\0');
    ConceptMake (StrC5, Ref5, C5, FALSE, '\0', '\0');
    ConceptMake (StrC6, Ref6, C6, FALSE, '\0', '\0');
    ConceptMake (StrC7, Ref7, C7, FALSE, '\0', '\0');
    C1->NextlnList = C2;
    C2->NextlnList = C3;
    C3->NextlnList = C4;
    C4->NextlnList = C5;
    C5->NextlnList = C6;
    C6->NextlnList = C7;
    MakeFiveDyadUnit (C1, C2, C3, C4);
    MakeFiveDyadUnit (C5, C6, C7, C4);

    Now copy the graph just made... the copy operation copies complex graphs, so we need to make a context for it first.
    
    ToBeCopied = typealloc (ContextListElt);
    ToBeCopiedCxt = typealloc (Context);
    ToBeCopiedSimpleGraphPtr = typealloc (ConceptListElt);

    ToBeCopied->NextlnList = ToBeCopiedCxt;
    ToBeCopied->NextlnList = 0;
    ToBeCopiedCxt->NextlnList = ToBeCopiedSimpleGraphPtr;
    ToBeCopiedSimpleGraphPtr->NextlnList = C1;
    ToBeCopiedSimpleGraphPtr->NextlnList = 0;

    start = times (&tmsstart);
    NewG = ShadowCopy (ToBeCopied, 0);
    end = times (&tmsendi);

    printf("...and display it, before exiting the program before anything else can happen."");
    
    printf("Test ID: COPY10\n\n");
    printf("Original Graph...\n\n");
    Graph_Display_Gateway (GRAPH, '\0', TypeDefs, RelDefs, FnDefs, ToBeCopied, CanonModel, TLattice, RLattice);
    printf("Copied Graph...\n\n");
void Copy5 (TLattice, RLattice, TypeDefs, RelDefs, FnDefs, CanonModel)
{
    LatticeElt *Lattice, *RLattice;
    Defn *TypeDefs, *RelDefs, *FnDefs;
    ContextListElt *CanonModel;
    struct tms start, end;
    struct tmsstart, tmsend;

    struct tmsstart, tmsend;
    clock_t start, end;
    ContextListElt *NewG, *ToBeCopied;
    Context *ToBeCopiedCxt;
    ConceptListElt *ToBeCopiedSimpleGraphPtr;
    Concept *C1, *C2, *C3, *C4;
    char *StrCl = "Cl", *StrC2 = "C2", *StrC3 = "C3", *StrC4 = "C4";
    char *Ref1 = "$1", *Ref2 = "$2", *Ref3 = "$3", *Ref4 = "$4";

    /******************************************************************
    /* This is Test Program COPY5 - testing speed of copying of a graph composed from 5 */
    /* dyads. It sets up a basic S-dyad graph cell and then copies it. */
    /******************************************************************/

    C1 = typealloc (Concept);
    C2 = typealloc (Concept);
    C3 = typealloc (Concept);
    C4 = typealloc (Concept);
    ConceptMake (StrC1, Ref1, C1, FALSE);
    ConceptMake (StrC2, Ref2, C2, FALSE);
    ConceptMake (StrC3, Ref3, C3, FALSE);
    ConceptMake (StrC4, Ref4, C4, FALSE);
    C1->Next = C2;
    C2->Next = C3;
    C3->Next = C4;
    MakeFiveDyadUnit (C1, C2, C3, C4);

    /******************************************************************
    /* Now copy the graph just made... the copy operation copies complex graphs, so we need to make a */
    /* context for it first. */
    /******************************************************************/

    ToBeCopied = typealloc (ContextListElt);
    ToBeCopiedCxt = typealloc (Context);
    ToBeCopiedSimpleGraphPtr = typealloc (ConceptListElt);
    ToBeCopied->PointedAt = ToBeCopiedCxt;
    ToBeCopied->NextlnList = 0;
    ToBeCopiedCxt->Depth = 0;
    ToBeCopiedCxt->ListOfMembers = ToBeCopiedSimpleGraphPtr;
    ToBeCopiedSimpleGraphPtr->PointedAt = C1;
    ToBeCopiedSimpleGraphPtr->NextlnList = 0;
    start = times (&tmsstart);
    NewG = ShadowCopy (ToBeCopied, 0);
    end = times (&tmsend);

    /******************************************************************
    /* ...and display it, before exiting the program before anything else can happen. */
    /******************************************************************/

    printf ("Test ID: COPY5\n\n");
    printf ("Original Graph...\n\n");
    Graph_Display_Gateway (GRAPH, \"\0\", TypeDefs, RelDefs, FnDefs, ToBeCopied, CanonModel, TLattice, RLattice);
    printf ("\n\n");
    printf ("Copied Graph...
\n");
    Graph_Display_Gateway (GRAPH, \"\0\", TypeDefs, RelDefs, FnDefs, NewG, CanonModel, TLattice, RLattice);
    printf ("\n\n");
    pr_times (end-start, &tmsstart, &tmsend);
}

void MakeTestLattices (TL, RL)
{
    LatticeElt *TL, *RL;
    
    LatticeEltPtr *MetaSPtr, *MetaNPtr, *MetaIPtr;
    char *StrCl = "C1\0", *StrC2 = "C2\0", *StrC3 = "C3\0", *StrC4 = "C4\0";
    char *StrR1 = "R1\0", *StrR2 = "R2\0", *StrR3 = "R3\0", *StrR4 = "R4\0", *StrR5 = "R5\0";
    char *String = "STRING\0", *Indv = "INDIVIDUAL\0", *Numb = "NUMERIC\0";

    /******************************************************************
    /* Simply makes the toy lattices: */
    /******************************************************************/
*/
UNIVERSAL >> STRING (UNIVERSAL) > (R1) */
UNIVERSAL >> NUMERIC (UNIVERSAL) > (R2) */
UNIVERSAL >> INDIVIDUAL (INDIVIDUAL) > (R3) */
INDIVIDUAL >> C1 (UNIVERSAL) > (R4) */
INDIVIDUAL >> C2 (UNIVERSAL) > (R5) */
INDIVIDUAL >> C3 (UNIVERSAL) */
INDIVIDUAL >> C4 (UNIVERSAL) */
*/
*/
*/
*/
*/
	for use in tests. */
	*****************************************************************************/

MetaS = typealloc (LatticeElt);
strcpy (MetaS->Label, '\0', 25);
strcpy (MetaS->Label, String, 7);
MetaS->SubLabels = 0;
MetaI = typealloc (LatticeElt);
strcpy (MetaI->Label, '\0', 25);
strcpy (MetaI->Label, Indiv, 11);
MetaI->SubLabels = 0;
MetaN = typealloc (LatticeElt);
strcpy (MetaN->Label, '\0', 25);
strcpy (MetaN->Label, Numb, 8);
MetaN->SubLabels = 0;
MetaSPtr = typealloc (LatticeEltPtr);
MetaIPtr = typealloc (LatticeEltPtr);
MetaNPtr = typealloc (LatticeEltPtr);
MetaSPtr->PointedAt = MetaS;
MetaSPtr->ArrType = ARISTO;
MetaSPtr->NextInList = MetaIPtr;
MetaIPtr->PointedAt = MetaI;
MetaIPtr->ArrType = ARISTO;
MetaIPtr->NextInList = MetaNPtr;
MetaNPtr->PointedAt = MetaN;
MetaNPtr->ArrType = ARISTO;
MetaNPtr->NextInList = 0;
TL->SubLabels = MetaSPtr;
C1 = typealloc (LatticeElt);
strcpy (C1->Label, '\0', 25);
strcpy (C1->Label, StrCl, 3);
C2 = typealloc (LatticeElt);
strcpy (C2->Label, '\0', 25);
strcpy (C2->Label, StrC2, 3);
C3 = typealloc (LatticeElt);
strcpy (C3->Label, '\0', 25);
strcpy (C3->Label, StrC3, 3);
C4 = typealloc (LatticeElt);
strcpy (C4->Label, '\0', 25);
strcpy (C4->Label, StrC4, 3);
C1Ptr = typealloc (LatticeEltPtr);
C1Ptr->PointedAt = C1;
C1Ptr->ArrType = ARISTO;
C2Ptr = typealloc (LatticeEltPtr);
C2Ptr->PointedAt = C2;
C2Ptr->ArrType = ARISTO;
C3Ptr = typealloc (LatticeEltPtr);
C3Ptr->PointedAt = C3;
C3Ptr->ArrType = ARISTO;
C4Ptr = typealloc (LatticeEltPtr);
C4Ptr->PointedAt = C4;
C4Ptr->ArrType = ARISTO;
MetaI->SubLabels = C1Ptr;
C1Ptr->NextInList = C2Ptr;
C2Ptr->NextInList = C3Ptr;
C3Ptr->NextInList = C4Ptr;
C4Ptr->NextInList = 0;
R1 = typealloc (LatticeElt);
strcpy (R1->Label, '\0', 25);
strcpy (R1->Label, StrR1, 3);
R2 = typealloc (LatticeElt);
strcpy (R2->Label, '\0', 25);
strcpy (R2->Label, StrR2, 3);
R3 = typealloc (LatticeElt);
strcpy (R3->Label, '\0', 25);
strcpy (R3->Label, StrR3, 3);
R4 = typealloc (LatticeElt);
strcpy (R4->Label, '\0', 25);
strcpy (R4->Label, StrR4, 3);
R5 = typealloc (LatticeElt);
strcpy (R5->Label, '\0', 25);
strcpy (R5->Label, StrR5, 3);
R1Ptr = typealloc (LatticeEltPtr);
R1Ptr->PointedAt = R1;
R1Ptr->ArrType = LOGICAL;
R2Ptr = typealloc (LatticeEltPtr);
R2Ptr->PointedAt = R2;
R2Ptr->ArcType = LOGICAL;
R3Ptr = typealloc (LatticeEltPtr);
R3Ptr->PointedAt = R3;
R3Ptr->ArcType = LOGICAL;
R4Ptr = typealloc (LatticeEltPtr);
R4Ptr->PointedAt = R4;
R4Ptr->ArcType = LOGICAL;
R5Ptr = typealloc (LatticeEltPtr);
R5Ptr->PointedAt = R5;
R5Ptr->ArcType = LOGICAL;
RL->SubLabels = R1Ptr;
R1Ptr->NextInList = R2Ptr;
R2Ptr->NextInList = R3Ptr;
R3Ptr->NextInList = R4Ptr;
R4Ptr->NextInList = R5Ptr;
R5Ptr->NextInList = 0;
}

void main ()
{
    Defn *TDL, *RDL, *FDL;
    LatticeElt *TLattice, *RLattice;
    ContextListElt *SOG;
    char *Nul = "\0", *StrU = "UNIVERSAL\0";
    TDL = typealloc (Defn);
    RDL = typealloc (Defn);
    FDL = typealloc (Defn);
    strncpy (TDL->DefName, Nul, 25);
    TDL->NextInList = 0;
    strncpy (RDL->DefName, Nul, 25);
    RDL->NextInList = 0;
    strncpy (FDL->DefName, Nul, 25);
    FDL->NextInList = 0;
    SOG = typealloc (ContextListElt);
    SOG->PointedAt = 0;
    SOG->NextInList = 0;
    TLattice = typealloc (LatticeElt);
    RLattice = typealloc (LatticeElt);
    strncpy (TLattice->Label, Nul, 25);
    strncpy (RLattice->Label, StrU, 9);
    TLattice->SubLabels = 0;
    strncpy (RLattice->Label, Nul, 25);
    strncpy (RLattice->Label, StrU, 9);
    RLattice->SubLabels = 0;
    MakeTestLattices (TLattice, RLattice);
    Copy5 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Copy10 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Copy20 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Copy40 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Copy80 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    exit(0);
}
This subappendix contains a printout of the routines used to test lattice searching in C-GRASS.

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include "cgiclass.h"

static void pr_times (real, tmsstart, tmson)
{
    long clktck;
    printf ("
    printf ("%7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf ("%7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf ("%7.7f\n", (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf ("%7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

int EndTest (Buff, StrLen)
{
    BufferUnit *Buff;
    int StrLen;
    if (StrLen < 2){
        return (FALSE);
    }
    if (Buff->Line[StrLen-2] == 'E'){
        if (Buff->Line[StrLen-1] == 'N'){
            if (Buff->Line[StrLen] == 'D'){
                return (TRUE);
            }
        }
    }
    else {
        return (FALSE);
    }
    if (Buff->Line[StrLen-3] == 'Q'
        & Buff->Line[StrLen-2] == 'U'
        & Buff->Line[StrLen-1] == 'I'
        & Buff->Line[StrLen] == 'T'){
        return (TRUE);
    }
    else {
        return (FALSE);
    }
}
```
`void MakeSubs (CurrDepth, BaseNode, MaxBushiness, LNamePtr)
int CurrDepth, MaxBushiness;
LatticeElt *BaseNode;
char LNNamePtr[25];
{
    int BushThisDepth, Bushiness, SubOne;
    LatticeElt *NewNode;
    LatticeEltPtr *FindEnd, *Nodeptr;
    BushThisDepth = rand() & MaxBushiness;
    if (BushThisDepth == 0){
        BushThisDepth = 1;
    }
    for (Bushiness = 1; Bushiness <= MaxBushiness; Bushiness++){
        NewNode = typealloc (LatticeElt);
        strncpy (NewNode->Label, '\0', 25);
        strcpy (NewNode->Label, MakeAlias (LNamePtr, 'N'));
        strncpy (LNamePtr, \0, 25);
        strncpy (LNamePtr, NewNode->Label, LongestString (NewNode->Label, '\0'));
        NewNode->SubLabels = 0;
        NodePtr = typealloc (LatticeEltPtr);
        NodePtr->PointedAt = NewNode;
        NodePtr->ArcType = ARISTO; /* Doesn't really matter for test purposes */
        NodePtr->NextInList = 0;
        if (BaseNode->SubLabels == 0){
            BaseNode->SubLabels = Nodeptr;
        } else {
            for (FindEnd = BaseNode->SubLabels; FindEnd->NextInList != 0; FindEnd = FindEnd->NextInList())
                FindEnd->NextInList = NodePtr;
        }
        SubOne = CurrDepth - 1;
        if (SubOne > 0){
            MakeSubs (SubOne, NewNode, MaxBushiness, LNamePtr);
        }
    }
}

void General_Lattice_Search (MaxDepth, MaxBushiness)
int MaxDepth, MaxBushiness;
{
    typedef struct {
        LatticeElt *CurrL;
        int Done;
        struct LList *NextInList;
    } LList;
    struct tms tmsstart, tmsend;
    clock_t start, end;
    LatticeElt *TestL, *TestNodePtr, *Subs;
    LatticeEltPtr *TestNodeSubs;
    int Hit, Found, StrL, NoAdd, Depth;
    char TOFind[25], LastName[25], *RootName = 'N1O';
    memcpy (LastName, '\0', 25);
    strncpy (LastName, RootName, 3);
    TestL = typealloc (LatticeElt);
    strncpy (TestL->Label, '\0', 25);
    TestL->SubLabels = 0;
    MakeSubs (MaxDepth, TestL, MaxBushiness, LastName);`
/* Print out hierarchy just generated. */

LListStub = typealloc(LList);
LListStub->CurrL = TestL;
LListStub->Done = FALSE;
LListStub->NextInList = 0;
Hit = FALSE;
printf ("Hierarchy Generated

-");
found = FALSE;
for (ScanLL = LListStub; ScanLL != 0 && found == FALSE; ScanLL = ScanLL->NextInList){
  if (ScanLL->Done == FALSE){
    found = TRUE;
    TestNodePtr = ScanLL->CurrL;
    for (TestNodeSubs = TestNodePtr->SubLabels; TestNodeSubs != 0; TestNodeSubs = TestNodeSubs->NextInList){
      Subs = TestNodeSubs->PointedAt;
      if (Subs->ArcType == LOGICAL){
        printf ("%s", LongestString (Subs->Label, ",", Subs->Label));
      }
      else {
        printf ("%s", LongestString (Subs->Label, ",", Subs->Label));
      }
      printf ("%s", Subs->Label);
    }
    NoAdd = FALSE;
    for (EndLL = LListStub; NoAdd == FALSE && EndLL != 0; EndLL = EndLL->NextInList){
      if (EndLL->CurrL == Subs){
        NoAdd = TRUE;
      }
    }
    if (NoAdd == FALSE){
      for (EndLL = LListStub; EndLL->NextInList != 0; EndLL = EndLL->NextInList){
        NewLL = typealloc(LList);
        NewLL->CurrL = Subs;
        NewLL->Done = FALSE;
        NewLL->NextInList = 0;
        EndLL->NextInList = NewLL;
      }
    }
    ScanLL->Done = TRUE;
  }
}
  if (found == FALSE){
    Hit = TRUE;
  }
} while (Hit == FALSE);
*/

strncpy (ToFind, ",", 25);
strncpy (ToFind, LastName, 25);

/* Now time lattice search. */

start = times (&tmsstart);
if (Sub (RootName, ToFind, TestL, FALSE, ANY) == TRUE){
  end = times (&tmsend);
  printf ("Depth %d, Maximum Bushiness %d in: %d\n", MaxDepth, MaxBushiness);
  printf ("Time to traverse lattice from %s to %s: %d\n", RootName, LastName);
  pr_times (end-start, &tmsstart, &tmsend);
}

void main ()
{
  General_Lattice_Search (2, 2);
  General_Lattice_Search (2, 4);
  General_Lattice_Search (2, 6);
  General_Lattice_Search (4, 2);
  General_Lattice_Search (4, 4);
  General_Lattice_Search (4, 6);
  General_Lattice_Search (6, 2);
  General_Lattice_Search (6, 4);
  General_Lattice_Search (6, 6);
  exit(0);
}
Appendix F (Subappendix D)
Posting Algorithm Tests

This subappendix contains two examples of the correctness of the posting algorithm of C-GRASS (these are the raw tests used in the discussion of the posting algorithm in Chapter 5), and an edited listing of the test programs used in this evaluation. First, the examples of correctness.

Creation Instruction: CREATE TABLE L72P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL, C3 CHAR(25) NOT NULL, C4 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L72P (C1, C2, C3, C4) SELECT L7P.C1, L7P.C2, L5P.C2, L6P.C2 FROM L7P, L5P, L6P WHERE L7P.C1 = L5P.C1 AND L7P.C1 = L6P.C1

Creation Instruction: CREATE TABLE L73P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L73P (C1, C2) SELECT L2P.C1, L2P.C2 FROM L2P

Creation Instruction: CREATE TABLE L74P (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL UNIQUE, C3 CHAR(25) NOT NULL)
Move Instruction: INSERT INTO L74P (C1, C2) SELECT L3P.C1, L3P.C2, L1P.C2 FROM L3P, L1P WHERE L3P.C2 = L1P.C1

Creation Instruction: CREATE TABLE L75P (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L75P (C1, C2) SELECT L8P.C1, L8P.C2 FROM L8P

Creation Instruction: CREATE TABLE L76N (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL, C3 CHAR(25) NOT NULL, C4 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L76N (C1, C2, C3, C4) SELECT L7N.C1, L7N.C2, L5N.C2, L6N.C2 FROM L7N, L5N, L6N WHERE L7N.C1 = L5N.C1 AND L7N.C1 = L6N.C1

Creation Instruction: CREATE TABLE L73N (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L73N (C1, C2) SELECT L2N.C1, L2N.C2 FROM L2N

Creation Instruction: CREATE TABLE L78N (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L78N (C1, C2) SELECT L10P.C1, L10P.C2 FROM L10P

Creation Instruction: CREATE TABLE L72N (C1 CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL, C3 CHAR(25) NOT NULL, C4 CHAR(25) NOT NULL UNIQUE)
Move Instruction: INSERT INTO L72N (C1, C2, C3, C4) SELECT L7N.C1, L7N.C2, L5N.C2, L6N.C2 FROM L7N, L5N, L6N WHERE L7N.C1 = L5N.C1 AND L7N.C1 = L6N.C1

Creation Instruction: CREATE TABLE L77N (C1 CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL UNIQUE, C3 CHAR(25) NOT NULL)
Move Instruction: INSERT INTO L77N (C1, C2) SELECT L8N.C1, L8N.C2 FROM L8N
Creation Instruction: CREATE TABLE L76N (Cl CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L76N (Cl, C2) SELECT L9N.Cl, L9N.C2 FROM L9N

Creation Instruction: CREATE TABLE L77N (Cl CHAR(25) NOT NULL, C2 CHAR(25) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L77N (Cl, C2) SELECT L4N.Cl, L4N.C2 FROM L4N

Creation Instruction: CREATE TABLE L78N (Cl CHAR(25) NOT NULL UNIQUE, C2 CHAR(25) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L78N (Cl, C2) SELECT LION.Cl, LION.C2 FROM LION

Creation Instruction: CREATE TABLE L82N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L82N (Cl, C2) SELECT LIP.Cl, LIP.C3 FROM LIP

Creation Instruction: CREATE TABLE L83N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L83N (Cl, C2) SELECT L2N.Cl, L2N.C2 FROM L2N

Creation Instruction: CREATE TABLE L84N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L84N (Cl, C2) SELECT LIN.Cl, LIN.C2 FROM LIN

Creation Instruction: CREATE TABLE L85N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L85N (Cl, C2) SELECT L3N.Cl, L3N.C2 FROM L3N

Creation Instruction: CREATE TABLE L86N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL, C3 CHAR(2S) NOT NULL UNIQUE, C4 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L86N (Cl, C2, C3, C4) SELECT L4N.Cl, L4N.C3, L4N.C4, L4N.C2 FROM L4N

Creation Instruction: CREATE TABLE L87N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L87N (Cl, C2) SELECT L6N.Cl, L6N.C2 FROM L6N

Creation Instruction: CREATE TABLE L88N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L88N (Cl, C2) SELECT L5N.Cl, L5N.C2 FROM L5N

Creation Instruction: CREATE TABLE L89N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L89N (Cl, C2) SELECT LSN.Cl, LSN.C2 FROM LSN

Creation Instruction: CREATE TABLE L90N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L90N (Cl, C2) SELECT L9N.Cl, L9N.C2 FROM L9N

Creation Instruction: CREATE TABLE L91N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L91N (Cl, C2) SELECT L7N.Cl, L7N.C2 FROM L7N

and:

Creation Instruction: CREATE TABLE L82P (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L82P (Cl, C2) SELECT L1P.C1, L1P.C3 FROM L1P

Creation Instruction: CREATE TABLE L83P (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L83P (Cl, C2) SELECT L2P.C1, L2P.C2 FROM L2P

Creation Instruction: CREATE TABLE L84P (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L84P (Cl, C2) SELECT L1P.C1, L1P.C2 FROM L1P

Creation Instruction: CREATE TABLE L85P (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L85P (Cl, C2) SELECT L3P.C1, L3P.C2 FROM L3P

Creation Instruction: CREATE TABLE L86P (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL, C3 CHAR(2S) NOT NULL UNIQUE, C4 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L86P (Cl, C2, C3, C4) SELECT L4P.C1, L4P.C3, L4P.C4, L4P.C2 FROM L4P

Creation Instruction: CREATE TABLE L87P (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L87P (Cl, C2) SELECT L6P.Cl, L6P.C2 FROM L6P

Creation Instruction: CREATE TABLE L88P (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L88P (Cl, C2) SELECT L5P.C1, L5P.C2 FROM L5P

Creation Instruction: CREATE TABLE L90P (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L90P (Cl, C2) SELECT L9P.Cl, L9P.C2 FROM L9P

Creation Instruction: CREATE TABLE L91P (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L91P (Cl, C2) SELECT L7P.C1, L7P.C2 FROM L7P

Creation Instruction: CREATE TABLE L82N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L82N (Cl, C2) SELECT L1N.Cl, L1N.C2 FROM L1N

Creation Instruction: CREATE TABLE L83N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L83N (Cl, C2) SELECT L2N.Cl, L2N.C2 FROM L2N

Creation Instruction: CREATE TABLE L84N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L84N (Cl, C2) SELECT L1N.Cl, L1N.C2 FROM L1N

Creation Instruction: CREATE TABLE L85N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL, UNIQUE (Cl, C2))
Move Instruction : INSERT INTO L85N (Cl, C2) SELECT L3N.Cl, L3N.C2 FROM L3N

Creation Instruction: CREATE TABLE L86N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL, C3 CHAR(2S) NOT NULL UNIQUE, C4 CHAR(2S) NOT NULL)
Move Instruction : INSERT INTO L86N (Cl, C2, C3, C4) SELECT L4N.C1, L4N.C3, L4N.C4, L4N.C2 FROM L4N

Creation Instruction: CREATE TABLE L87N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L87N (Cl, C2) SELECT L6N.Cl, L6N.C2 FROM L6N

Creation Instruction: CREATE TABLE L88N (Cl CHAR(2S) NOT NULL, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L88N (Cl, C2) SELECT L5N.Cl, L5N.C2 FROM L5N

Creation Instruction: CREATE TABLE L90N (Cl CHAR(2S) NOT NULL UNIQUE, C2 CHAR(2S) NOT NULL UNIQUE)
Move Instruction : INSERT INTO L90N (Cl, C2) SELECT L9N.Cl, L9N.C2 FROM L9N
Note that the raw output shown above also includes instructions to restructure negative as well as positive graphs; these were omitted from the discussion of the algorithm in Chapter 5.

Secondly, an edited version of the test programs (we have trimmed the second posting test):

```
EXEC SQL INCLUDE SQLCA;
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include <cgcclass.h>
static void pr_times (real, tmsstart, tmsend)
clock_t real;
struct tms *tmsstart, *tmsend;
static long clktck = 0;
/***************************/
/* Timing Display Routine. */
/***************************/
clktck = sysconf (_SC_CLK_TCK);
printf("%7.7f\n", real / (double) clktck);
printf("%7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
printf("%7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
user: %7.7f
"
,(tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
sys: %7.7f
"
(tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
user: %7.7£


/* Simply makes these toy lattice for testing: */
/ * UNIVERSAL >> STRING (UNIVERSAL) > (R1) * /
/ * UNIVERSAL >> NUMERIC (UNIVERSAL) > (R2) * /
/ * UNIVERSAL >> INDIVIDUAL (UNIVERSAL) > (R3) * /
/ * INDIVIDUAL >> JOB (UNIVERSAL) > (R4) * /
/ * INDIVIDUAL >> PERSON (UNIVERSAL) > (R5) * /
/ * INDIVIDUAL >> DEPT */
```
MetaS = typealloc (LatticeElt);
strncpy (MetaS->Label, '0', 25);
strncpy (MetaS->Label. String, 7);
MetaS->SubLabels = 0;
MetaI = typealloc (LatticeElt);
strncpy (MetaI->Label, '0', 25);
strncpy (MetaI->Label. Indv, 11);
MetaI->SubLabels = 0;
MetaN = typealloc (LatticeElt);
strncpy (MetaN->Label, '0', 25);
strncpy (MetaN->Label. Name, 8);
MetaI->SubLabels = 0;
MetaSPtr = typealloc (LatticeEltPtr);
MetaIPtr = typealloc (LatticeEltPtr);
MetaNPtr = typealloc (LatticeEltPtr);
MetaSPtr->ArcType = ARISTO;
MetaSPtr->NextInList = MetaIPtr;
MetaIPtr->PointedAt = MetaS;
MetaSPtr->ArcType = ARISTO;
MetaIPtr->NextInList = MetaNPtr;
MetaNPtr->PointedAt = MetaN;
MetaNPtr->ArcType = ARISTO;
MetaNPtr->NextInList = 0;
TL->SubLabels = MetaSPtr;

C1 = typealloc (LatticeElt);
strncpy (C1->Label, '0', 25);
strncpy (C1->Label, FName, 5);
C1->SubLabels = 0;
C2 = typealloc (LatticeElt);
strncpy (C2->Label, '0', 25);
strncpy (C2->Label, JobName, 7);
C2->SubLabels = 0;
C3 = typealloc (LatticeElt);
strncpy (C3->Label, '0', 25);
strncpy (C3->Label, ISBN, 8);
C3->SubLabels = 0;
C4 = typealloc (LatticeElt);
strncpy (C4->Label, '0', 25);
strncpy (C4->Label, CourseName, 10);
C4->SubLabels = 0;
C5 = typealloc (LatticeElt);
strncpy (C5->Label, '0', 25);
strncpy (C5->Label, ProjName, 8);
C5->SubLabels = 0;
C6 = typealloc (LatticeElt);
strncpy (C6->Label, '0', 25);
strncpy (C6->Label, ISHN, 4);
C6->SubLabels = 0;
C7 = typealloc (LatticeElt);
strncpy (C7->Label, '0', 25);
strncpy (C7->Label, LibName, 7);
C7->SubLabels = 0;
C8 = typealloc (LatticeElt);
strncpy (C8->Label, '0', 25);
strncpy (C8->Label, BookTitle, 9);
C8->SubLabels = 0;
C9 = typealloc (LatticeElt);
strncpy (C9->Label, '0', 25);
strncpy (C9->Label, PubName, 9);
C9->SubLabels = 0;
C10 = typealloc (LatticeElt);
strncpy (C10->Label, '0', 25);
strncpy (C10->Label, AuthName, 9);
C10->SubLabels = 0;
C11 = typealloc (LatticeElt);
strncpy (C11->Label, '0', 25);
strncpy (C11->Label, Hours, 5);
C11->SubLabels = 0;
C12 = typealloc (LatticeElt);
strncpy (C12->Label, '0', 25);
strncpy (C12->Label, YearPub, 8);
C12->SubLabels = 0;
C13 = typealloc (LatticeElt);
strncpy (C13->Label, 'Job', 3);
C13->SubLabels = 0;
C14 = typealloc (LatticeElt);
strncpy (C14->Label, 'Person', 6);
C14->SubLabels = 0;
C15 = typealloc (LatticeElt);
strncpy (C15->Label, 'Dept', 4);
C15->SubLabels = 0;
C16 = typealloc (LatticeElt);
strncpy (C16->Label, 'Course', 6);
C16->SubLabels = 0;
C17 = typealloc (LatticeElt);
strncpy (C17->Label, 'ProjRole', 8);
C17->SubLabels = 0;
C18 = typealloc (LatticeElt);
strncpy (C18->Label, 'Project', 7);
C18->SubLabels = 0;
C19 = typealloc (LatticeElt);
strncpy (C19->Label, 'Budget', 6);
C19->SubLabels = 0;
C20 = typealloc (LatticeElt);
strncpy (C20->Label, 'CourseDate', 10);
C20->SubLabels = 0;
C21 = typealloc (LatticeElt);
strncpy (C21->Label, 'Copy', 4);
C21->SubLabels = 0;
C22 = typealloc (LatticeElt);
strncpy (C22->Label, 'Place', 5);
C22->SubLabels = 0;
C23 = typealloc (LatticeElt);
strncpy (C23->Label, 'PT', 2);
C23->SubLabels = 0;
C24 = typealloc (LatticeElt);
strncpy (C24->Label, 'CompanyEntity', 11);
C24->SubLabels = 0;
C1Pcr = typealloc (LatticeEltptr);
C2Pcr = typealloc (LatticeEltptr);
C3Pcr = typealloc (LatticeEltptr);
C4Pcr = typealloc (LatticeEltptr);
C5Pcr = typealloc (LatticeEltptr);
C6Pcr = typealloc (LatticeEltptr);
C7Pcr = typealloc (LatticeEltptr);
C8Pcr = typealloc (LatticeEltptr);
C9Pcr = typealloc (LatticeEltptr);
C10Pcr = typealloc (LatticeEltptr);
C11Pcr = typealloc (LatticeEltptr);
C12Pcr = typealloc (LatticeEltptr);
C13Pcr = typealloc (LatticeEltptr);
C14Pcr = typealloc (LatticeEltptr);
C15Pcr = typealloc (LatticeEltptr);
C16Pcr = typealloc (LatticeEltptr);
C17Pcr = typealloc (LatticeEltptr);
C18Pcr = typealloc (LatticeEltptr);
C19Pcr = typealloc (LatticeEltptr);
C20Pcr = typealloc (LatticeEltptr);
C21Pcr = typealloc (LatticeEltptr);
C22Pcr = typealloc (LatticeEltptr);
C23Pcr = typealloc (LatticeEltptr);
C24Pcr = typealloc (LatticeEltptr);
C1Pcr->PointedAt = C1;
C1Pcr->ArcType = ARISTO;
C1Pcr->NextInList = C2Pcr;
C2Pcr->PointedAt = C2;
C2Pcr->ArcType = ARISTO;
C2Pcr->NextInList = C3Pcr;
C3Pcr->PointedAt = C3;
C3Pcr->ArcType = ARISTO;
C3Pcr->NextInList = C4Pcr;
C4Pcr->PointedAt = C4;
C4Pcr->ArcType = ARISTO;
C4Pcr->NextInList = C5Pcr;
C5Pcr->PointedAt = C5;
C5Pcr->ArcType = ARISTO;
C5Ptr->NextInList = C6Ptr;
C6Ptr->PointedAt = C6;
C6Ptr->ArcType = ARISTO;
C6Ptr->NextInList = C7Ptr;
C7Ptr->PointedAt = C7;
C7Ptr->ArcType = ARISTO;
C7Ptr->NextInList = C8Ptr;
C8Ptr->PointedAt = C8;
C8Ptr->ArcType = ARISTO;
C8Ptr->NextInList = C9Ptr;
C9Ptr->PointedAt = C9;
C9Ptr->ArcType = ARISTO;
C9Ptr->NextInList = C10Ptr;
C10Ptr->PointedAt = C10;
C10Ptr->ArcType = ARISTO;
C10Ptr->NextInList = 0;
MetaS->SubLabels = C1Ptr;
C11Ptr->PointedAt = C11;
C11Ptr->ArcType = ARISTO;
C11Ptr->NextInList = C12Ptr;
C12Ptr->PointedAt = C12;
C12Ptr->ArcType = ARISTO;
C12Ptr->NextInList = 0;
MetaN->SubLabels = C11Ptr;
C13Ptr->PointedAt = C13;
C13Ptr->ArcType = ARISTO;
C13Ptr->NextInList = C14Ptr;
C14Ptr->PointedAt = C14;
C14Ptr->ArcType = ARISTO;
C14Ptr->NextInList = C15Ptr;
C15Ptr->PointedAt = C15;
C15Ptr->ArcType = ARISTO;
C15Ptr->NextInList = C16Ptr;
C16Ptr->PointedAt = C16;
C16Ptr->ArcType = ARISTO;
C16Ptr->NextInList = C17Ptr;
C17Ptr->PointedAt = C17;
C17Ptr->ArcType = ARISTO;
C17Ptr->NextInList = C18Ptr;
C18Ptr->PointedAt = C18;
C18Ptr->ArcType = ARISTO;
C18Ptr->NextInList = C19Ptr;
C19Ptr->PointedAt = C19;
C19Ptr->ArcType = ARISTO;
C19Ptr->NextInList = C20Ptr;
C20Ptr->PointedAt = C20;
C20Ptr->ArcType = ARISTO;
C20Ptr->NextInList = C21Ptr;
C21Ptr->PointedAt = C21;
C21Ptr->ArcType = ARISTO;
C21Ptr->NextInList = C22Ptr;
C22Ptr->PointedAt = C22;
C22Ptr->ArcType = ARISTO;
C22Ptr->NextInList = C23Ptr;
C23Ptr->PointedAt = C23;
C23Ptr->ArcType = ARISTO;
C23Ptr->NextInList = C24Ptr;
C24Ptr->PointedAt = C24;
C24Ptr->ArcType = ARISTO;
C24Ptr->NextInList = 0;
MetaL->SubLabels = C24Ptr;
R1 = typealloc (LatticeElt);
strcpy (R1->Label, '0', 25);
strcpy (R1->Label, StrR1, 3);
R1->SubLabels = 0;
R2 = typealloc (LatticeElt);
strcpy (R2->Label, '0', 25);
strcpy (R2->Label, StrR2, 3);
R2->SubLabels = 0;
R3 = typealloc (LatticeElt);
strcpy (R3->Label, '0', 25);
strcpy (R3->Label, StrR3, 3);
R3->SubLabels = 0;
R4 = typealloc (LatticeElt);
strcpy (R4->Label, '0', 25);
strcpy (R4->Label, StrR4, 3);
R4->SubLabels = 0;
R5 = typealloc (LatticeElt);
strcpy (R5->Label, '0', 25);
strcpy (R5->Label, StrR5, 3);
R5->SubLabels = 0;
R1Ptr = typealloc (LatticeEltPtr);
R2Ptr = typealloc (LatticeEltPtr);
R3Ptr = typealloc (LatticeEltPtr);
void PostingTest() (TLatt, FOG, FDL)
LatticeElt *TLatt;
ContextListElt *FOG;
void *FDL;

struct timespec tsstart, tsend;
clock_t start, end;
char *St1 = "1", *St2 = "2", *St3 = "3", *St4 = "4", *St5 = "5", *St6 = "6", *St7 = "7", *St8 = "8", *St9 = "9";
char *St10 = "10", *St11 = "11";
char *Copy = "COPY", *Place = "PLACE", *LibName = "LIBNAME", *PT = "PT", *BookTitle = "BOOKTITLE", *CompanyEntity = "COMPANY_ENT";
*ISBN = "ISBN", *YearPub = "YEAR_PUBL", *person = "PERSON", *PubName = "PUBL_NAME";
char *Loc = "LOC", *Name = "NAME", *Attr = "ATTR";

//This is POST_1.PC, the first test of posting and scheme design. The initial list of /
// dependency and membership considerations are as follow: */
/*
/*[COPY: *1] -> (LOC) -> [PLACE: *2]
/*[PLACE: *2] -> (NAME) -> [LIBNAME: *3]
/*[PUBLISHED-THING: *4] -> (ATTR) -> [COPY: *1]
/*[PUBLISHED-THING: *4] -> (ATTR) -> (YEAR-PUBLISHED: *8)
/*[PUBLISHED-THING: *4] -> (ATTR) -> (PERSON: *9)
/*[PERSON: *9] -> (NAME) -> [AUTHOR-NAME: *10]
/*
/*All of these are optional N-M optional. The tables baed on this list are:
/*
/* L1 (COPY PLACE) L6 (PUBLISHED-THING ISBN)
/* L2 (PLACE LIBNAME) L7 (PUBLISHED-THING YEAR-PUBLISHED)
/* L3 (PUBLISHED-THING COPY) L8 (PUBLISHED-THING PERSON)
/* L4 (PUBLISHED-THING BOOKTITLE) L9 (PERSON AUTHOR-NAME)
/* L5 (PUBLISHED-THING COMPANY-ENTITY) L10 (COMPANY-ENTITY PUBLISHER-NAME)
/*
/*
/* 01 = typealloc (Cardinality);
/* 02 = typealloc (Cardinality);
/* 03 = typealloc (Cardinality);
/* 04 = typealloc (Cardinality);
/* 05 = typealloc (Cardinality);
/* 06 = typealloc (Cardinality);
/* 07 = typealloc (Cardinality);
/* 08 = typealloc (Cardinality);
/* 09 = typealloc (Cardinality);
/* 10 = typealloc (Cardinality);
/* strncpy (01->Relation, "0", 25);
/* strncpy (02->Relation, Loc, 3);
/* strncpy (03->FromPos, "0", 25);
/* strncpy (04->FromPos, Copy, 4);
/* strncpy (05->FPtr, "4", 25);
/* strncpy (06->FPtr, St1, 2);
/* strncpy (07->ToPos, "0", 25);
/* strncpy (08->ToPos, Place, 5);
/* strncpy (09->TPtr, "0", 25);
/* strncpy (10->TPtr, St2, 2);
/* strncpy (01->FromTab, "0", 25);
/* strncpy (01->FromTab, Li, 2);
Ol->FromDom = 1;
strncpy (Ol->ToTab, '\0', 25);
strncpy (Ol->ToTab, L1, 2);
Ol->ToDom = 2;
Ol->FromMembership = OPTIONAL;
Ol->ToMembership = OPTIONAL;
Ol->Cardinality = MANY_MANY;
Ol->Ignore = FALSE;
Ol->NextInList = 0;
strncpy (Ol->Relation, '\0', 25);
strncpy (Ol->Relation, Name, 4);
strncpy (Ol->FromPos, '\0', 25);
strncpy (Ol->ToPos, Place, 5);
strncpy (Ol->FRef, '\0', 25);
strncpy (Ol->TRef, St2, 2);
strncpy (Ol->ToPos, '0', 25);
strncpy (Ol->ToPos, LibName, 7);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, St3, 2);
strncpy (Ol->FromTab, '\0', 25);
strncpy (Ol->FromTab, L2, 2);
Ol->FromDom = 1;
strncpy (Ol->ToTab, '\0', 25);
strncpy (Ol->ToTab, L2, 2);
Ol->ToDom = 2;
Ol->FromMembership = OPTIONAL;
Ol->ToMembership = OPTIONAL;
Ol->Cardinality = MANY_MANY;
Ol->Ignore = FALSE;
Ol->NextInList = 0;
strncpy (Ol->Relation, '\0', 25);
strncpy (Ol->Relation, Attr, 4);
strncpy (Ol->FromPos, '\0', 25);
strncpy (Ol->FromPos, PT, 2);
strncpy (Ol->FRef, '\0', 25);
strncpy (Ol->FRef, St4, 2);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, Copy, 4);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, St1, 2);
strncpy (Ol->FromTab, '\0', 25);
strncpy (Ol->FromTab, L3, 2);
Ol->FromDom = 1;
strncpy (Ol->ToTab, '\0', 25);
strncpy (Ol->ToTab, L3, 2);
Ol->ToDom = 2;
Ol->FromMembership = OPTIONAL;
Ol->ToMembership = OPTIONAL;
Ol->Cardinality = MANY_MANY;
Ol->Ignore = FALSE;
Ol->NextInList = 0;
strncpy (Ol->Relation, '\0', 25);
strncpy (Ol->Relation, Name, 4);
strncpy (Ol->FromPos, '\0', 25);
strncpy (Ol->FromPos, PT, 2);
strncpy (Ol->FRef, '\0', 25);
strncpy (Ol->FRef, St4, 2);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, BookTitle, 7);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, St5, 2);
strncpy (Ol->FromTab, '\0', 25);
strncpy (Ol->FromTab, L4, 2);
Ol->FromDom = 1;
strncpy (Ol->ToTab, '\0', 25);
strncpy (Ol->ToTab, L4, 2);
Ol->ToDom = 2;
Ol->FromMembership = OPTIONAL;
Ol->ToMembership = OPTIONAL;
Ol->Cardinality = MANY_MANY;
Ol->Ignore = FALSE;
Ol->NextInList = 0;
strncpy (Ol->Relation, '\0', 25);
strncpy (Ol->Relation, Attr, 4);
strncpy (Ol->FromPos, '\0', 25);
strncpy (Ol->FromPos, PT, 2);
strncpy (Ol->FRef, '\0', 25);
strncpy (Ol->FRef, St4, 2);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, CompanyEntity, 11);
strncpy (Ol->TRef, '0', 25);
strncpy (Ol->TRef, St6, 2);
strncpy (Ol->FromTab, '\0', 25);
strncpy (Ol->FromTab, L5, 2);
Ol->FromDom = 1;
strncpy (Ol->ToTab, '\0', 25);
strncpy (Ol->ToTab, L5, 2);
Ol->ToDom = 2;
05->From_Membership = OPTIONAL;
05->To_Membership = OPTIONAL;
05->Cardinality = MANY_MANY;
05->Ignore = FALSE;
05->NextInList = 06;
strncpy (06->Relation, '\0', 25);
strncpy (06->Relation, Attr_4);
strncpy (06->FromPos, '\0', 25);
strncpy (06->FromPos, PT_2);
strncpy (06->FPRef, '\0', 25);
strncpy (06->FPRef, St4_2);
strncpy (06->ToPos, '\0', 25);
strncpy (06->ToPos, ISBN_4);
strncpy (06->TPRef, '\0', 25);
strncpy (06->TPRef, St7_2);
strncpy (06->FromTab, '\0', 25);
strncpy (06->FromTab, L6_2);
06->FromDom = 1;
strncpy (06->ToTab, '\0', 25);
strncpy (06->ToTab, L6_2);
06->ToDom = 2;
06->From_Membership = OPTIONAL;
06->To_Membership = OPTIONAL;
06->Cardinality = MANY_MANY;
06->Ignore = FALSE;
06->NextInList = 07;
strncpy (07->Relation, '\0', 25);
strncpy (07->Relation, Attr_4);
strncpy (07->FromPos, '\0', 25);
strncpy (07->FromPos, PT_2);
strncpy (07->FPRef, '\0', 25);
strncpy (07->FPRef, St4_2);
strncpy (07->ToPos, YearPub_8);
strncpy (07->TPRef, '\0', 25);
strncpy (07->TPRef, St8_2);
strncpy (07->FromTab, '\0', 25);
strncpy (07->FromTab, L7_2);
07->FromDom = 1;
strncpy (07->ToTab, '\0', 25);
strncpy (07->ToTab, L7_2);
07->ToDom = 2;
07->From_Membership = OPTIONAL;
07->To_Membership = OPTIONAL;
07->Cardinality = MANY_MANY;
07->Ignore = FALSE;
07->NextInList = 08;
strncpy (08->Relation, '\0', 25);
strncpy (08->Relation, Attr_8);
strncpy (08->FromPos, '\0', 25);
strncpy (08->FromPos, PT_2);
strncpy (08->FPRef, '\0', 25);
strncpy (08->FPRef, St4_2);
strncpy (08->ToPos, '\0', 25);
strncpy (08->ToPos, Person_6);
strncpy (08->TPRef, '\0', 25);
strncpy (08->TPRef, St9_2);
strncpy (08->FromTab, '\0', 25);
strncpy (08->FromTab, L8_2);
08->FromDom = 1;
strncpy (08->ToTab, '\0', 25);
strncpy (08->ToTab, L8_2);
08->ToDom = 2;
08->From_Membership = OPTIONAL;
08->To_Membership = OPTIONAL;
08->Cardinality = MANY_MANY;
08->Ignore = FALSE;
08->NextInList = 09;
strncpy (09->Relation, '\0', 25);
strncpy (09->Relation, Name_4);
strncpy (09->FromPos, '\0', 25);
strncpy (09->FromPos, Person_6);
strncpy (09->FPRef, '\0', 25);
strncpy (09->FPRef, St9_2);
strncpy (09->ToPos, '\0', 25);
strncpy (09->ToPos, AuthName_9);
strncpy (09->TPRef, '\0', 25);
strncpy (09->TPRef, St10_3);
strncpy (09->FromTab, '\0', 25);
strncpy (09->FromTab, L9_2);
09->FromDom = 1;
strncpy (09->ToTab, '\0', 25);
strncpy (09->ToTab, L9_2);
09->ToDom = 2;
09->From_Membership = OPTIONAL;
09->To_Membership = OPTIONAL;
09->Cardinality = MANY_MANY;
09->Ignore = FALSE;
09->NextInList = 04;
strncpy (01->Relation, 'L0', 25);
strncpy (01->Relation, Name, 4);
strncpy (01->FromPos, '0', 25);
strncpy (01->FromPos, CompanyEntity, 11);
strncpy (01->FPRef, St6, 2);
strncpy (01->ToPos, '0', 25);
strncpy (01->ToPos, PubName, 9);
strncpy (01->TPRef, '0', 25);
strncpy (01->TPRef, St11, 3);
strncpy (01->FromTab, '0', 25);
strncpy (01->FromTab, L10, 3);
01->FromDom = 1;
strncpy (01->ToTab, '0', 25);
strncpy (01->ToTab, L10, 3);
01->ToDom = 2;
01->From_Membership = OPTIONAL;
01->To_Membership = OPTIONAL;
01->Cardinality = MANY_MANY;
01->Ignore = FALSE;
01->NextInList = 05;

N1 = typealloc (Cardinality);
N2 = typealloc (Cardinality);
N3 = typealloc (Cardinality);
N4 = typealloc (Cardinality);
N5 = typealloc (Cardinality);
N6 = typealloc (Cardinality);
N7 = typealloc (Cardinality);
N8 = typealloc (Cardinality);
N9 = typealloc (Cardinality);
N10 = typealloc (Cardinality);

strncpy (N1->Relation, '0', 25);
strncpy (N1->Relation, Loc, 5);
strncpy (N1->FromPos, '0', 25);
strncpy (N1->FromPos, Copy, 4);
strncpy (N1->FPRef, '0', 25);
strncpy (N1->ToPos, '0', 25);
strncpy (N1->ToPos, Place, 5);
strncpy (N1->TPRef, St, 2);
strncpy (N1->TPRef, St2, 2);
strncpy (N1->FromTab, '0', 25);
strncpy (N1->FromTab, L1, 2);
N1->FromDom = 1;
strncpy (N1->ToTab, '0', 25);
strncpy (N1->ToTab, L1, 2);
N1->ToDom = 2;
N1->From_Membership = OBLIGATORY;
N1->To_Membership = OPTIONAL;
N1->Cardinality = MANY_ONE;
N1->Ignore = FALSE;
N1->NextInList = N6;
strncpy (N2->Relation, '0', 25);
strncpy (N2->Relation, Name, 4);
strncpy (N2->FromPos, '0', 25);
strncpy (N2->FromPos, Place, 5);
strncpy (N2->FPRef, '0', 25);
strncpy (N2->FPRef, St, 2);
strncpy (N2->ToPos, '0', 25);
strncpy (N2->ToPos, PubName, 7);
strncpy (N2->ToPos, LibName, 7);
strncpy (N2->FromTab, '0', 25);
strncpy (N2->FromTab, St3, 2);
strncpy (N2->FromTab, L2, 2);
N2->FromDom = 1;
strncpy (N2->ToTab, '0', 25);
strncpy (N2->ToTab, L2, 2);
N2->ToDom = 2;
N2->From_Membership = OBLIGATORY;
N2->To_Membership = OBLIGATORY;
strncpy (N7->Relation, Attr, 4);
strncpy (N7->FromPos, '\0', 25);
strncpy (N7->FromPos, PT, 2);
strncpy (N7->FPRef, '\0', 25);
strncpy (N7->FPRef, St4, 2);
strncpy (N7->ToPos, YearPub, 8);
strncpy (N7->TPRef, '\0', 25);
strncpy (N7->TPRef, St8, 2);
strncpy (N7->FromTab, '\0', 25);
strncpy (N7->FromTab, L7, 2);
N7->FromDom = 1;
strncpy (N7->ToTab, '\0', 25);
strncpy (N7->ToTab, L7, 2);
N7->ToDom = 2;
N7->From_Membership = OBLIGATORY;
N7->To_Membership = OPTIONAL;
N7->Cardinality = MANY_ONE;
N7->Ignore = FALSE;
N7->NextInList = N2;
strncpy (N8->Relation, Attr, 4);
strncpy (N8->FromPos, Person, 6);
strncpy (N8->FromPos, St9, 2);
strncpy (N8->ToPos, Person, 6);
strncpy (N8->ToPos, PubName, 9);
strncpy (N8->TPRef, St10, 3);
strncpy (N8->TPRef, St9, 2);
strncpy (N8->FromTab, '\0', 25);
strncpy (N8->FromTab, L8, 2);
N8->FromDom = 1;
strncpy (N8->ToTab, '\0', 25);
strncpy (N8->ToTab, L8, 2);
N8->ToDom = 2;
N8->From_Membership = OBLIGATORY;
N8->To_Membership = OBLIGATORY;
N8->Cardinality = MANY_MANY;
N8->Ignore = FALSE;
N8->NextInList = N3;
strncpy (N9->Relation, '\0', 25);
strncpy (N9->Relation, AuthName, 9);
strncpy (N9->FromPos, PubName, 9);
strncpy (N9->FromPos, PubName, 9);
strncpy (N9->FromPos, AuthName, 9);
strncpy (N9->FromPos, St10, 3);
strncpy (N9->FromPos, St11, 3);
strncpy (N9->FromPos, St10, 3);
N9->FromDom = 1;
strncpy (N9->ToTab, '\0', 25);
strncpy (N9->ToTab, L9, 2);
N9->ToDom = 2;
N9->From_Membership = OBLIGATORY;
N9->To_Membership = OBLIGATORY;
N9->Cardinality = ONE_ONE;
N9->Ignore = FALSE;
N9->NextInList = N4;
strncpy (N10->Relation, '\0', 25);
strncpy (N10->Relation, CompanyEntity, 11);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, CompanyEntity, 11);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, PubName, 9);
strncpy (N10->FromPos, '\0', 25);
strncpy (N10->FromPos, '\0', 25);
N10->FromDom = 1;
strncpy (N10->ToTab, '\0', 25);
strncpy (N10->ToTab, L10, 3);
N10->ToDom = 2;
N10->From_Membership = OBLIGATORY;
N10->To_Membership = OBLIGATORY;
N10->Cardinality = ONE_ONE;
N10->Ignore = FALSE;
N10->NextInList = N5;

start = times (&tmsstart);
CompareCardLists (01, N7, TLatt, FOG, FDL);
end = times (&tmsend);
printf ("\nTime taken (seconds) to perform Posting Test \
\n");
void main ()
{
    Defn *FDL;
    LatticeElt *TLattice, *RLattice;
    ContextListElt *FOG;
    char *Nul = "\0", *StrU = "UNIVERSAL\0";
    EXEC SQL BEGIN DECLARE SECTION;
    char "Slash = ";
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT :Slash:
    FDL = typealloc (Defn);
    strncpy (FDL->DefName, Nul, 25);
    FDL->NextInList = 0;
    FOG = typealloc (ContextListElt);
    FOG->PointedAt = 0;
    FOG->NextInList = 0;
    TLattice = typealloc (LatticeElt);
    strncpy (TLattice->Label, Nul, 25);
    strncpy (TLattice->Label, StrU, 9);
    TLattice->SubLabels = 0;
    strncpy (RLattice->Label, Nul, 25);
    strncpy (RLattice->Label, StrU, 9);
    RLattice->SubLabels = 0;
    MakeTestLattices (TLattice, RLattice);
    PostingTest1 (TLattice, FOG, FDL);
    PostingTest2 (TLattice, FOG, FDL);
    EXEC SQL COMMIT WORK RELEASE;
    exit (0);
}
Appendix F (Subappendix E)

Projection Tests

This subappendix contains a heavily edited listing of the test programs used to test projection in C-GRASS (as most of the test routines look very similar, we have included a number of the lesser tests only).

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include <cgclass.h>

static void pr_times (real, trnsstart, tmsend)
{
  clock_t real;
  struct tms *tmsstart, *tmsend;
  long clktck;
  
  clktck = sysconf (_SC_CLK_TCK);
  printf("%7.7f\n", real / (double) clktck);
  printf("%7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
  printf("%7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
  printf("%7.7f\n", (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
  printf("%7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

void MakeFiveDyadUnit (Cl, C2, C3, C4)
{
  Concept *Cl, *C2, *C3, *C4;
  char StrR1[10], StrR2[10], StrR3[10], StrR4[10], StrR5[10];
  R1 = typealloc (Relation);
  R2 = typealloc (Relation);
  R3 = typealloc (Relation);
  R4 = typealloc (Relation);
  R5 = typealloc (Relation);
  RelationMake (StrR1, R1);
  RelationMake (StrR2, R2);
  RelationMake (StrR3, R3);
  RelationMake (StrR4, R4);
  RelationMake (StrR5, R5);
```

RelationMake (StrR4, R4);
RelationMake (StrR5, R5);
AddRNode (C1, R1, FROM, '\0', 0);
AddRNode (C2, R1, TO, '\0', 0);
AddRNode (C3, R2, FROM, '\0', 0);
AddRNode (C4, R2, TO, '\0', 0);
AddRNode (C5, R3, FROM, '\0', 0);
AddRNode (C6, R3, TO, '\0', 0);
AddRNode (C7, R4, FROM, '\0', 0);
AddRNode (C8, R4, TO, '\0', 0);
AddRNode (C9, R5, FROM, '\0', 0);
AddRNode (C10, R5, TO, '\0', 0);
}

void Projection_1020 (TLattice, RLattice, TypeDefs, RelDefs, FnDefs, CanonModel)

LatticeElt *TLattice, *RLattice;
TypeDef *TypeDefs, *RelDefs, *FnDefs;
ContextListElt *CanonModel;

struct tms tmsstart, tmsend;
clock_t start, end;
GraphElt *GraphsFound, *LoopAround, *ScanThrough;
MapElt *MapS_LTo_v;
ContextListElt *Complex_u, *Complex_v, *CurrG;
Context *Complex_uCtx, *Complex_vCtx, *CurrGC;
ConceptListElt *comple_u_SimpleGraphPtr, *Comple_v_SimpleGraphPtr,
*ListOfProjections, *LoopMemb;
Concept *Clu, *C2u, *C3u, *C4u, *C5u, *C6u, *C7u,
*Clv, *C2v, *C3v, *C4v, *C5v, *C6v, *C7v, *C8v,
RelationListElt *LoopRL;
Relation *LoopRel;
int *Success, Succ;
char *StrCl, *StrC2, *StrC3, *StrC4,
*StrC5, *StrC6, *StrC7, *StrC8,
*StrC9, *StrC10, *StrC11, *StrC12;
char *Ref1u, *Ref1v, *Ref2u, *Ref2v,
*Ref3u, *Ref3v, *Ref4u, *Ref4v,
*Ref5u, *Ref5v, *Ref6u, *Ref6v,
*Ref7u, *Ref7v, *Ref8u, *Ref8v,
*Ref9u, *Ref9v, *Ref10u, *Ref10v;

/****************************************************/
/* This is Test Program PJ_1020 - testing the projection of u = 10 dyads in v = 20 dyads.*/
/****************************************************/

Clu = typealloc (Concept);
C2u = typealloc (Concept);
C3u = typealloc (Concept);
C4u = typealloc (Concept);
C5u = typealloc (Concept);
C6u = typealloc (Concept);
C7u = typealloc (Concept);
ConceptMake (StrCl, Ref1u, Clu, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref2u, C2u, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref3u, C3u, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref4u, C4u, FALSE, '\0', '\0');
ConceptMake (StrC5, Ref5u, C5u, FALSE, '\0', '\0');
ConceptMake (StrC6, Ref6u, C6u, FALSE, '\0', '\0');
ConceptMake (StrC7, Ref7u, C7u, FALSE, '\0', '\0');
Clu->Next = C2u;
C2u->Next = C3u;
C3u->Next = C4u;
C4u->Next = C5u;
C5u->Next = C6u;
C6u->Next = C7u;
MakeFiveDyadUnit (Clu, C2u, C3u, C4u);
MakeFiveDyadUnit (C5u, C6u, C7u, C4u);

/*****************************
/* Make up graph v here (20 dyads). */
/*****************************

Clv = typealloc (Concept);
C2v = typealloc (Concept);
C3v = typealloc (Concept);
C4v = typealloc (Concept);
C5v = typealloc (Concept);
C6v = typealloc (Concept);
C7v = typealloc (Concept);
C8v = typealloc (Concept);
C9v = typealloc (Concept);
C10v = typealloc (Concept);
C11v = typealloc (Concept);
C12v = typealloc (Concept);
ConceptMake (StrCl, Ref1v, Clv, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref2v, C2v, FALSE, '\0', '\0');
ConceptMake (StrC3, Ref3v, C3v, FALSE, '\0', '\0');
ConceptMake (StrC4, Ref4v, C4v, FALSE, '\0', '\0');
ConceptMake (StrC5, Ref5v, C5v, FALSE, '\0', '\0');
ConceptMake (StrC6, Ref6v, C6v, FALSE, '\0', '\0');
ConceptMake (StrC7, Ref7v, C7v, FALSE, '\0', '\0');
ConceptMake (StrC8, Ref8v, C8v, FALSE, '\0', '\0');
ConceptMake (StrC9, Ref9v, C9v, FALSE, '\0', '\0');
ConceptMake (StrC10, Ref10v, C10v, FALSE, '\0', '\0');
ConceptMake (StrC11, Ref11v, C11v, FALSE, '\0', '\0');
ConceptMake (StrC12, Ref12v, C12v, FALSE, '\0', '\0');
ConceptMake (StrC2, Ref6v, C6v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC3, Ref7v, C7v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC4, Ref8v, C8v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC5, Ref9v, C9v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC6, Ref10v, C10v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC7, Ref11v, C11v, FALSE, \"\0\", \"\0\");
ConceptMake (StrC8, Ref12v, C12v, FALSE, \"\0\", \"\0\");

Clv->Next = C2v;
C2v->Next = C3v;
C3v->Next = C4v;
C4v->Next = C5v;
C5v->Next = C6v;
C6v->Next = C7v;
C7v->Next = C8v;
C8v->Next = C9v;
C9v->Next = C10v;
C10v->Next = C11v;
C11v->Next = C12v;

MakeFiveDyadUnit (Clv, C2v, C3v, C4v);
MakeFiveDyadUnit (C5v, C6v, C7v, C4v);
MakeFiveDyadUnit (C9v, C8v, C3v, C10v);
MakeFiveDyadUnit (C12v, C11v, C7v, C10v);

/****
* Now project u into v here. *
/****

Complex_u = typealloc (ContextListElt);
Complex_u_Cxt = typealloc (Context);
Complex_u_SimpleGraphPtr = typealloc (ConceptListElt);
Complex_u->PointedAt = Complex_u_Cxt;
Complex_u->NextInList = 0;
Complex_u->Used = 'N';
Complex_u->Ident = NOTYETDEF;
Complex_u_Cxt->Depth = 0;
Complex_u_Cxt->Dominating = 0;
Complex_u_Cxt->Processed = NOTYETDEF;
Complex_u_Cxt->ListOfMembers = Complex_u_SimpleGraphPtr;
Complex_u_SimpleGraphPtr->PointedAt = Clu;
Complex_u_SimpleGraphPtr->Used = 'N';
Complex_u_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_u_SimpleGraphPtr->NextInList = 0;
Complex_v = typealloc (ContextListElt);
Complex_v_Cxt = typealloc (Context);
Complex_v_SimpleGraphPtr = typealloc (ConceptListElt);
Complex_v->PointedAt = Complex_v_Cxt;
Complex_v->NextInList = 0;
Complex_v->Used = 'N';
Complex_v->Ident = NOTYETDEF;
Complex_v_Cxt->Depth = 0;
Complex_v_Cxt->Dominating = 0;
Complex_v_Cxt->Processed = NOTYETDEF;
Complex_v_Cxt->ListOfMembers = Complex_v_SimpleGraphPtr;
Complex_v_SimpleGraphPtr->PointedAt = Clv;
Complex_v_SimpleGraphPtr->Used = 'N';
Complex_v_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_v_SimpleGraphPtr->NextInList = 0;

Succ = TRUE;
Success = &Succ;
Maps_u_To_v = typealloc (MapElt);
start = times (&tmsstart);
GraphsFound = Complex_Projection_Gateway (Complex_u, Complex_v, TLattice, RLattice, Success, Maps_u_To_v, FALSE, FALSE, FALSE);
end = times (&tmsend);
for (LoopAround = GraphsFound; LoopAround != 0; LoopAround = LoopAround->NextInList) {
    CurrG = LoopAround->ComplexGraph;
    CurrGC = CurrG->PointedAt;
    for (LoopMemb = CurrGC->ListOfMembers; LoopMemb != 0; LoopMemb = LoopMemb->NextInList) {
        for (LoopConc = LoopMemb->PointedAt; LoopConc != 0; LoopConc = LoopConc->Next) {
            LoopRel = LoopConc->RelationList;
            if (LoopRel != 0) {
                LoopRel->Done = 'N';
            }
        }
    }
}

print ("Graph u...

n");
This is Test Program PJ.0505 - testing the projection of \( u = 5 \) dyads in \( v = 5 \) dyads. This program sets up the two graphs \( u \) and \( v \) (which in this case are identical, and so there should be a single projection if working correctly), then submits them to the projection routine of C-GRASS and displays the result. The graph \( u \) (and also \( v \)) has the form:

\[
\begin{align*}
\text{C1: } & \text{[a]} - \text{(R1)->[C2: } \text{[b]} - \\
& \text{[C3: } \text{[c]} - \text{(R4)->[C4: } \text{[d]} - \\
& \text{(R3)->[C1: } \text{[a]}].
\end{align*}
\]

This graph has five dyads.

A few notes on the routines used. The typealloc function is a C-GRASS function that allocates the necessary space for a particular object; ConceptMake simply makes a concept, given an object, label and referent, etc.

The first step is to make graph \( u \)... out of a basic unit of four concepts and five relations...
C3v->Next = C4v;
MakeFiveDyadUnit (Clv, C2v, C3v, C4v);

/******************************************************************************
/* Now project u into v here. */
/******************************************************************************
Complex_u = typealloc (ContextListElt);
Complex_u_Cxt = typealloc (Context);
Complex_u_SimpleGraphPtr = typealloc (ConceptListElt);
Complex_u->pointedAt = Complex_u_Cxt;
Complex_u->NextInList = 0;
Complex_u->Used = 'N';
Complex_u->Ident = NOTYETDEF;
Complex_u_Cxt->Depth = 0;
Complex_u_Cxt->Dominating = 0;
Complex_u_Cxt->Processed = NOTYETDEF;
Complex_u_Cxt->ListOfMembers = Complex_u_SimpleGraphPtr;
Complex_u_SimpleGraphPtr->pointedAt = Complex_u;
Complex_u_SimpleGraphPtr->Used = 'N';
Complex_u_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_u_SimpleGraphPtr->NextInList = 0;
Complex_v = typealloc (ContextListElt);
Complex_v_Cxt = typealloc (Context);
Complex_v_SimpleGraphPtr = typealloc (ConceptListElt);
Complex_v->pointedAt = Complex_v_Cxt;
Complex_v->NextInList = 0;
Complex_v->Used = 'N';
Complex_v->Ident = NOTYETDEF;
Complex_v_Cxt->Depth = 0;
Complex_v_Cxt->Dominating = 0;
Complex_v_Cxt->Processed = NOTYETDEF;
Complex_v_Cxt->ListOfMembers = Complex_v_SimpleGraphPtr;
Complex_v_SimpleGraphPtr->pointedAt = Complex_v;
Complex_v_SimpleGraphPtr->Used = 'N';
Complex_v_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_v_SimpleGraphPtr->NextInList = 0;
Succ = TRUE;
Success = &Succ;
Maps_u_to_v = typealloc (MapElt);
start = times (&tmsstart);
GraphsFound = Complex_Projection_Gateway (Complex_u, Complex_v, TLattice, RLattice, Success, Maps_u_to_v, FALSE, FALSE, FALSE);
end = times (&tmsend);
for (LoopAround = GraphsFound; LoopAround != 0; LoopAround = LoopAround->NextInList){
  CurrG = LoopAround->ComplexGraph;
  CurrGC = CurrG->pointedAt;
  for (LoopMemb = CurrGC->ListOfMembers; LoopMemb != 0; LoopMemb = LoopMemb->NextInList){
    LoopConc = LoopMemb->pointedAt;
    for (LoopRL = LoopConc->RelationList; LoopRL != 0; LoopRL = LoopRL->NextInList){
      LoopRel = LoopRL->pointedAt;
      if (LoopRel != 0){
        LoopRel->Done = 'N';
      }
    }
  }
}

/********************************************************************************
/* Display the projections and exit the program before anything else can happen. */
/********************************************************************************

printf (-Graph u

-I;
Graph_Display_Gateway (GRAPH, '0', TypeDefs, RelDefs, FnDefs, Complex_u, CanonModel, TLattice, RLattice);
printf (-Graph v

-I;
Graph_Display_Gateway (GRAPH, '0', TypeDefs, RelDefs, FnDefs, Complex_v, CanonModel, TLattice, RLattice);
printf ('\n\nProjections of u in v ...

-);
Graph_Display_Gateway (GRAPH, '0', TypeDefs, RelDefs, FnDefs, ScanThrough->ComplexGraph, CanonModel, TLattice, RLattice);
pr_times (end-start, &tmsstart, &tmsend);
}
void MakeTestLattices (TL, RL)
LatticeElt *TL, *RL;
LatticeEltPtr *MetaSPtr, *MetaIPtr, *MetaNPtr;
char *StrCl = "Cl\0", *StrC2 = "C2\0", *StrC3 = "C3\0", *StrC4 = "C4\0";
char *StrR1 = "R1\0", *StrR2 = "R2\0", *StrR3 = "R3\0", *StrR4 = "R4\0", *StrR5 = "R5\0";
char *String = "STRING\0", *Indv = "INDIVIDUAL\0", *Numb = "NUMERIC\0";

/****** Simply makes the toy lattices: ******/
/****** UNIVERSAL >> STRING  (UNIVERSAL) > (R1) ******
/****** UNIVERSAL >> NUMERIC  (UNIVERSAL) > (R2) ******
/****** UNIVERSAL >> INDIVIDUAL (UNIVERSAL) > (R3) ******
/****** INDIVIDUAL >> C1  (UNIVERSAL) > (R4) ******
/****** INDIVIDUAL >> C2  (UNIVERSAL) > (R5) ******
/****** INDIVIDUAL >> C3 ******
/****** INDIVIDUAL >> C4 ******
******
/****** for use in tests. ******/
/******

MetaS = typealloc (LatticeElt);
strncpy (MetaS->Label, '"', 25);
strncpy (MetaS->Label, String, 7);
MetaS->SubLabels = 0;
MetaI = typealloc (LatticeElt);
strncpy (MetaI->Label, '"', 25);
strncpy (MetaI->Label, Indv, 11);
MetaI->SubLabels = 0;
MetaN = typealloc (LatticeElt);
strncpy (MetaN->Label, '"', 25);
strncpy (MetaN->Label, Numb, 8);
MetaN->SubLabels = 0;
MetaSPtr = typealloc (LatticeEltPtr);
MetaIPtr = typealloc (LatticeEltPtr);
MetaNPtr = typealloc (LatticeEltPtr);
MetaSPtr->PointedAt = MetaS;
MetaSPtr->ArcType = ARISTO;
MetaSPtr->NextInList = MetaIPtr;
MetaIPtr->PointedAt = MetaI;
MetaIPtr->ArcType = ARISTO;
MetaIPtr->NextInList = MetaNPtr;
MetaNPtr->PointedAt = MetaN;
MetaNPtr->ArcType = ARISTO;
MetaNPtr->NextInList = 0;
TL->SubLabels = MetaSPtr;

Cl = typealloc (LatticeElt);
strncpy (Cl->Label, '"', 25);
strncpy (Cl->Label, StrCl, 3);
C2 = typealloc (LatticeElt);
strncpy (C2->Label, '"', 25);
strncpy (C2->Label, StrC2, 3);
C3 = typealloc (LatticeElt);
strncpy (C3->Label, '"', 25);
strncpy (C3->Label, StrC3, 3);
C4 = typealloc (LatticeElt);
strncpy (C4->Label, '"', 25);
strncpy (C4->Label, StrC4, 3);

CIPtr = typealloc (LatticeEltPtr);
CIPtr->PointedAt = Cl;
CIPtr->ArcType = ARISTO;
C2Ptr = typealloc (LatticeEltPtr);
C2Ptr->PointedAt = C2;
C2Ptr->ArcType = ARISTO;
C3Ptr = typealloc (LatticeEltPtr);
C3Ptr->PointedAt = C3;
C3Ptr->ArcType = ARISTO;
C4Ptr = typealloc (LatticeEltPtr);
C4Ptr->PointedAt = C4;
C4Ptr->ArcType = ARISTO;
MetaI->SubLabels = CIPtr;
CIPtr->NextInList = C2Ptr;
C2Ptr->NextInList = C3Ptr;
C3Ptr->NextInList = C4Ptr;
C4Ptr->NextInList = 0;

R1 = typealloc (LatticeElt);
strncpy (R1->Label, '"', 25);
strncpy (R1->Label, StrR1, 3);
R2 = typealloc (LatticeElt);
strncpy (R2->Label, '"', 25);
strncpy (R2->Label, StrR2, 3);
R3 = typealloc (LatticeElt);
strncpy (R3->Label, '"', 25);
strncpy (R3->Label, StrR3, 3);
R4 = typealloc (LatticeElt);
strncpy (R4->Label, '"', 25);
void main()
{
    Defn *TDL, *RDL, *FDL;
    LatticeElt *TLattice, *RLattice;
    ContextListElt *SOG;
    char *Nul = \"\0\", *StrU = \"UNIVERSAL\0\";
    TDL = typealloc (Defn);
    RDL = typealloc (Defn);
    FDL = typealloc (Defn);
    strncpy (TDL->DefName, Nul, 25);
    TDL->NextInList = 0;
    strncpy (RDL->DefName, Nul, 25);
    RDL->NextInList = 0;
    strncpy (FDL->DefName, Nul, 25);
    FDL->NextInList = 0;
    SOG = typealloc (ContextListElt);
    SOG->PointedAt = 0;
    SOG->NextInList = 0;
    TLattice = typealloc (LatticeElt);
    RLattice = typealloc (LatticeElt);
    strncpy (TLattice->Label, Nul, 25);
    TLattice->SubLabels = 0;
    strncpy (RLattice->Label, StrU, 9);
    RLattice->SubLabels = 0;
    MakeTestLattices (TLattice, RLattice);
    Projection_0505 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_0510 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_0520 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_1000 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_1005 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_1010 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_1020 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_1040 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_2000 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_4000 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_4010 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_4020 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_6000 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_6010 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_6020 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_6040 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    Projection_6060 (TLattice, RLattice, TDL, RDL, FDL, SOG);
    exit(0);
}
Appendix F (Subappendix F)

\( \phi_\Delta \) Tests

This subappendix contains a heavily edited listing of the test programs used to test projection in C-GRASS (as most of the test routines look very similar, we have included a number of the lesser tests only - the tests where \( \nu = 80 \) are large, and have been deleted here).

```c
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include "cgclass.h"

static void pr_times (real, tmsstart, tmsend)
    clock_t real;
    struct tms *tmsstart, *tmsend;
{
    static long clktck = 0;
    
    printf("%7.7f\n", real / (double) clktck);
    printf("%7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("%7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("%7.7f\n", (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf("%7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

void MakeTestLattices (TL, RL)
{
    LatticeEltptr *MetaSPtr, *MetaIPtr, *MetaNPtr;
    char *StrCl = "Cl", *StrC2 = "C2", *StrC3 = "C3", *StrC4 = "C4", *StrR1 = "R1", *StrR2 = "R2", *StrR3 = "R3", *StrR4 = "R4", *StrRS = "RS", *String = "STRING";
    char *Indv = "INDIVIDUAL", *Numb = "NUMERIC";

    MetaS = typealloc (LatticeEltptr);
    strncpy (MetaS->Label, \',\', 25);
    strncpy (MetaS->SubLabels, 0);
    MetaI = typealloc (LatticeEltptr);
    strncpy (MetaI->Label, \',\', 25);

    /* Simply makes the toy lattices: */
    /* */
    /* UNIVERSAL >> STRING */
    /* UNIVERSAL >> NUMERIC */
    /* UNIVERSAL >> INDIVIDUAL */
    /* INDIVIDUAL >> C1 */
    /* INDIVIDUAL >> C2 */
    /* INDIVIDUAL >> C3 */
    /* INDIVIDUAL >> C4 */
    /* */
    /* for use in tests. */
```
void Phi_Delta_Operate (DyadsInGraph, NumberOfSubGraphs)
{
    struct tm *start, *end;
    ContentListElit *CList;
    RestrictionListElit *RList;
    JoinListElit *JList;
    Defn *TDL, *RDL, *FDL;
    LatticeElit *TLattice, *RLattice;
    ContextListElit *Complex, *Scrub;
    RestrictionListElit *ComplexElt, *ScrubElt;
    ConceptListElit *Comple~SirnpleGraphPtr, *ScrubMemb;
    int CF, JF, RF, *CFlag, *JFlag, *RFlag;
    char *StrCI = "CI", *StrC2 = "C2", *StrC3 = "C3", *StrC4 = "C4", *StrC5 = "C5";
    char *StrR1 = "R1", *StrR2 = "R2", *StrR3 = "R3", *StrR4 = "R4", *StrR5 = "R5";
    char *StrL1 = "L1", *StrL2 = "L2", *StrL3 = "L3", *StrL4 = "L4", *StrL5 = "L5";
    char *StrRef1 = "Ref1", *StrRef2 = "Ref2", *StrRef3 = "Ref3", *StrRef4 = "Ref4";
    char *StrUT = "UNIVERSAL";
    char *InitTab[25], *IT = "TO", *LastView = "v0";
    switch (DyadsInGraph){
    case 10:
        CI = typealloc (Concept);
        C2 = typealloc (Concept);
        C3 = typealloc (Concept);
        C4 = typealloc (Concept);
        C5 = typealloc (Concept);
        C6 = typealloc (Concept);
        C7 = typealloc (Concept);
        TDL = typealloc (Defn);
        RDL = typealloc (Defn);
        FDL = typealloc (Defn);
        strncpy (TDL->DefName, Nul, 25);
        strncpy (RDL->DefName, Nul, 25);
        strncpy (FDL->DefName, Nul, 25);
        TLattice = typealloc (LatticeElt);
        RLattice = typealloc (LatticeElt);
        strncpy (TLattice->Label, Nul, 25);
        strncpy (RLattice->Label, Nul, 25);
        strncpy (TLattice->SubLabels, Nul, 25);
        strncpy (RLattice->SubLabels, Nul, 25);
        MakeTestLattices (TLattice, RLattice);
        switch (DyadsInGraph){
        case 10:
            CI = typealloc (Concept);
            C2 = typealloc (Concept);
            C3 = typealloc (Concept);
            C4 = typealloc (Concept);
            C5 = typealloc (Concept);
            C6 = typealloc (Concept);
            C7 = typealloc (Concept);
switch (NumberOfSubGraphs)
{
    case 10:
        AddRNode (Cl, Rl, FROM, L1, 1);
        AddRNode (C2, R1, TO, L1, 2);
        AddRNode (C3, R2, FROM, L2, 1);
        AddRNode (C4, R2, TO, L2, 2);
        AddRNode (Cl, R3, FROM, L3, 1);
        AddRNode (C2, R3, TO, L3, 2);
        AddRNode (C3, R4, FROM, L4, 1);
        AddRNode (C4, R4, TO, L4, 2);
        AddRNode (C5, R5, FROM, L5, 1);
        AddRNode (C6, R5, TO, L5, 2);
        AddRNode (C7, R6, FROM, L1, 1);
        AddRNode (C8, R6, TO, L1, 2);
        AddRNode (C9, R7, FROM, L2, 1);
        AddRNode (C10, R7, TO, L2, 2);
        AddRNode (C11, R8, FROM, L3, 1);
        AddRNode (C12, R8, TO, L3, 2);
        AddRNode (C13, R9, FROM, L4, 1);
        AddRNode (C14, R9, TO, L4, 2);
        AddRNode (C15, R10, FROM, L5, 1);
        AddRNode (C16, R10, TO, L5, 2);
        break;
    case 20:
        Cl = typealloc (Concept);
        C2 = typealloc (Concept);
        C3 = typealloc (Concept);
        C4 = typealloc (Concept);
        C5 = typealloc (Concept);
        C6 = typealloc (Concept);
        C7 = typealloc (Concept);
        C8 = typealloc (Concept);
        C9 = typealloc (Concept);
        C10 = typealloc (Concept);
        C11 = typealloc (Concept);
        C12 = typealloc (Concept);
        ConceptMake (StrCl, Ref1, Cl, FALSE, '\0', '\0');
        ConceptMake (StrC2, Ref2, C2, FALSE, '\0', '\0');
        ConceptMake (StrC3, Ref3, C3, FALSE, '\0', '\0');
        ConceptMake (StrC4, Ref4, C4, FALSE, '\0', '\0');
        ConceptMake (StrC5, Ref5, C5, FALSE, '\0', '\0');
        ConceptMake (StrC6, Ref6, C6, FALSE, '\0', '\0');
        ConceptMake (StrC7, Ref7, C7, FALSE, '\0', '\0');
        ConceptMake (StrC8, Ref8, C8, FALSE, '\0', '\0');
        ConceptMake (StrC9, Ref9, C9, FALSE, '\0', '\0');
        ConceptMake (StrC10, Ref10, Cl0, FALSE, '\0', '\0');
        ConceptMake (StrC11, Ref11, Cl1, FALSE, '\0', '\0');
        ConceptMake (StrC12, Ref12, C12, FALSE, '\0', '\0');
        C1->Next = C2;
        C2->Next = C3;
        C3->Next = C4;
        C4->Next = C5;
        C5->Next = C6;
        C6->Next = C7;
        C7->Next = C8;
C8->Next = C9;
C9->Next = C10;
C10->Next = C11;
C11->Next = C12;
R1 = typealloc (Relation);
R2 = typealloc (Relation);
R3 = typealloc (Relation);
R4 = typealloc (Relation);
R5 = typealloc (Relation);
R6 = typealloc (Relation);
R7 = typealloc (Relation);
R8 = typealloc (Relation);
R9 = typealloc (Relation);
R10 = typealloc (Relation);
R11 = typealloc (Relation);
R12 = typealloc (Relation);
R13 = typealloc (Relation);
R14 = typealloc (Relation);
R15 = typealloc (Relation);
R16 = typealloc (Relation);
R17 = typealloc (Relation);
R18 = typealloc (Relation);
R19 = typealloc (Relation);
R20 = typealloc (Relation);
RelationMake (StrR1, R1);
RelationMake (StrR2, R2);
RelationMake (StrR3, R3);
RelationMake (StrR4, R4);
RelationMake (StrR5, R5);
RelationMake (StrR6, R6);
RelationMake (StrR7, R7);
RelationMake (StrR8, R8);
RelationMake (StrR9, R9);
RelationMake (StrR10, R10);
RelationMake (StrR11, R11);
RelationMake (StrR12, R12);
RelationMake (StrR13, R13);
RelationMake (StrR14, R14);
RelationMake (StrR15, R15);
RelationMake (StrR16, R16);
RelationMake (StrR17, R17);
RelationMake (StrR18, R18);
RelationMake (StrR19, R19);
RelationMake (StrR20, R20);
switch (NumberOfSubGraphs) {
  case 10:
    AddRNode (C1, R1, FROM, L1, 1);
    AddRNode (C2, R1, TO, L1, 2);
    AddRNode (C3, R3, FROM, L1, 1);
    AddRNode (C4, R3, TO, L1, 3);
    AddRNode (C2, R5, FROM, L1, 2);
    AddRNode (C4, R5, TO, L1, 3);
    AddRNode (C5, R6, FROM, L1, 1);
    AddRNode (C6, R6, TO, L1, 2);
    AddRNode (C5, R8, FROM, L1, 1);
    AddRNode (C4, R8, TO, L1, 3);
    AddRNode (C6, R10, FROM, L1, 2);
    AddRNode (C6, R10, TO, L1, 3);
    AddRNode (C7, R17, FROM, L2, 1);
    AddRNode (C6, R17, TO, L2, 2);
    AddRNode (C3, R4, FROM, L3, 1);
    AddRNode (C4, R4, TO, L3, 2);
    AddRNode (C7, R9, FROM, L3, 3);
    AddRNode (C4, R9, TO, L3, 2);
    AddRNode (C9, R11, FROM, L1, 1);
    AddRNode (C8, R11, TO, L1, 1);
    AddRNode (C9, R13, FROM, L1, 1);
    AddRNode (C10, R13, TO, L1, 3);
    AddRNode (C8, R15, FROM, L1, 2);
    AddRNode (C10, R15, TO, L1, 3);
    AddRNode (C3, R12, FROM, L2, 1);
    AddRNode (C8, R12, TO, L2, 2);
    AddRNode (C12, R16, FROM, L1, 1);
    AddRNode (C11, R16, TO, L1, 2);
    AddRNode (C12, R18, FROM, L1, 1);
    AddRNode (C10, R18, TO, L1, 3);
    AddRNode (C11, R20, FROM, L1, 2);
    AddRNode (C10, R20, TO, L1, 3);
    AddRNode (C7, R7, FROM, L2, 1);
    AddRNode (C11, R7, TO, L2, 2);
    AddRNode (C7, R19, FROM, L3, 1);
    AddRNode (C10, R19, TO, L3, 2);
    AddRNode (C3, R14, FROM, L3, 1);
    AddRNode (C10, R14, TO, L3, 2);
    break;
  case 20:
    AddRNode (C1, R1, FROM, L1, 1);
    AddRNode (C2, R1, TO, L1, 2);
AddRNode (C3, R2, FROM, L2, 1);
AddRNode (C2, R2, TO, L2, 2);
AddRNode (C1, R3, FROM, L3, 1);
AddRNode (C4, R3, TO, L3, 2);
AddRNode (C3, R4, FROM, L4, 1);
AddRNode (C4, R4, TO, L4, 2);
AddRNode (C2, R5, FROM, L5, 1);
AddRNode (C4, R5, TO, L5, 1);
AddRNode (C5, R6, FROM, L1, 1);
AddRNode (C6, R6, TO, L1, 2);
AddRNode (C7, R7, FROM, L2, 1);
AddRNode (C6, R7, TO, L2, 2);
AddRNode (C5, R8, FROM, L3, 1);
AddRNode (C4, R8, TO, L3, 2);
AddRNode (C7, R9, FROM, L4, 1);
AddRNode (C4, R9, TO, L4, 2);
AddRNode (C6, R10, FROM, L5, 1);
AddRNode (C4, R10, TO, L5, 2);
AddRNode (C9, R11, FROM, L1, 1);
AddRNode (C8, R11, TO, L1, 2);
AddRNode (C3, R12, FROM, L2, 1);
AddRNode (C8, R12, TO, L2, 2);
AddRNode (C9, R13, FROM, L3, 1);
AddRNode (C10, R13, TO, L3, 2);
AddRNode (C3, R14, FROM, L4, 1);
AddRNode (C10, R14, TO, L4, 2);
AddRNode (C8, R15, FROM, L5, 1);
AddRNode (C4, R15, TO, L5, 1);
AddRNode (C5, R16, FROM, L1, 1);
AddRNode (C6, R16, TO, L1, 2);
AddRNode (C7, R17, FROM, L2, 1);
AddRNode (C6, R17, TO, L2, 2);
AddRNode (C5, R18, FROM, L3, 1);
AddRNode (C10, R18, TO, L3, 2);
AddRNode (C7, R19, FROM, L4, 1);
AddRNode (C10, R19, TO, L4, 2);
AddRNode (C5, R20, FROM, L5, 1);
AddRNode (C4, R20, TO, L5, 2);
break;
ocase 40:
C1 = typealloc (Concept);
C2 = typealloc (Concept);
C3 = typealloc (Concept);
C4 = typealloc (Concept);
C5 = typealloc (Concept);
C6 = typealloc (Concept);
C7 = typealloc (Concept);
C8 = typealloc (Concept);
C9 = typealloc (Concept);
C10 = typealloc (Concept);
C11 = typealloc (Concept);
C12 = typealloc (Concept);
C13 = typealloc (Concept);
C14 = typealloc (Concept);
C15 = typealloc (Concept);
C16 = typealloc (Concept);
C17 = typealloc (Concept);
C18 = typealloc (Concept);
C19 = typealloc (Concept);
C20 = typealloc (Concept);
C21 = typealloc (Concept);
C22 = typealloc (Concept);
ConceptMake (StrC1, Ref1, C1, FALSE, '0', '0');
ConceptMake (StrC2, Ref2, C2, FALSE, '0', '0');
ConceptMake (StrC3, Ref3, C3, FALSE, '0', '0');
ConceptMake (StrC4, Ref4, C4, FALSE, '0', '0');
ConceptMake (StrC5, Ref5, C5, FALSE, '0', '0');
ConceptMake (StrC2, Ref6, C6, FALSE, '0', '0');
ConceptMake (StrC3, Ref7, C7, FALSE, '0', '0');
ConceptMake (StrC2, Ref8, C8, FALSE, '0', '0');
ConceptMake (StrC1, Ref9, C9, FALSE, '0', '0');
ConceptMake (StrC4, Ref10, C10, FALSE, '0', '0');
ConceptMake (StrC2, Ref11, C11, FALSE, '0', '0');
ConceptMake (StrC1, Ref12, C12, FALSE, '0', '0');
ConceptMake (StrC1, Ref13, C13, FALSE, '0', '0');
ConceptMake (StrC4, Ref14, C14, FALSE, '0', '0');
ConceptMake (StrC3, Ref15, C15, FALSE, '0', '0');
ConceptMake (StrC4, Ref16, C16, FALSE, '0', '0');
ConceptMake (StrC1, Ref17, C17, FALSE, '0', '0');
ConceptMake (StrC1, Ref18, C18, FALSE, '0', '0');
ConceptMake (StrC2, Ref19, C19, FALSE, '0', '0');
ConceptMake (StrC3, Ref20, C20, FALSE, '0', '0');
ConceptMake (StrC2, Ref21, C21, FALSE, '0', '0');
ConceptMake (StrC1, Ref22, C22, FALSE, '0', '0');
C1->Next = C2;
C2->Next = C3;
C3->Next = C4;
C4->Next = C5;
C5->Next = C6;
C6->Next = C7;
C7->Next = C8;
C8->Next = C9;
C9->Next = C10;
C10->Next = C11;
C11->Next = C12;
C12->Next = C13;
C13->Next = C14;
C14->Next = C15;
C15->Next = C16;
C16->Next = C17;
C17->Next = C18;
C18->Next = C19;
C19->Next = C20;
C20->Next = C21;
C21->Next = C22;
R1 = typealloc (Relation);
R2 = typealloc (Relation);
R3 = typealloc (Relation);
R4 = typealloc (Relation);
R5 = typealloc (Relation);
R6 = typealloc (Relation);
R7 = typealloc (Relation);
R8 = typealloc (Relation);
R9 = typealloc (Relation);
R10 = typealloc (Relation);
R11 = typealloc (Relation);
R12 = typealloc (Relation);
R13 = typealloc (Relation);
R14 = typealloc (Relation);
R15 = typealloc (Relation);
R16 = typealloc (Relation);
R17 = typealloc (Relation);
R18 = typealloc (Relation);
R19 = typealloc (Relation);
R20 = typealloc (Relation);
R21 = typealloc (Relation);
R22 = typealloc (Relation);
R23 = typealloc (Relation);
R24 = typealloc (Relation);
R25 = typealloc (Relation);
R26 = typealloc (Relation);
R27 = typealloc (Relation);
R28 = typealloc (Relation);
R29 = typealloc (Relation);
R30 = typealloc (Relation);
R31 = typealloc (Relation);
R32 = typealloc (Relation);
R33 = typealloc (Relation);
R34 = typealloc (Relation);
R35 = typealloc (Relation);
R36 = typealloc (Relation);
R37 = typealloc (Relation);
R38 = typealloc (Relation);
R39 = typealloc (Relation);
R40 = typealloc (Relation);
RelationMake (StrR1, R1);
RelationMake (StrR2, R2);
RelationMake (StrR3, R3);
RelationMake (StrR4, R4);
RelationMake (StrR5, R5);
RelationMake (StrR6, R6);
RelationMake (StrR7, R7);
RelationMake (StrR8, R8);
RelationMake (StrR9, R9);
RelationMake (StrR10, R10);
RelationMake (StrR11, R11);
RelationMake (StrR12, R12);
RelationMake (StrR13, R13);
RelationMake (StrR14, R14);
RelationMake (StrR15, R15);
RelationMake (StrR16, R16);
RelationMake (StrR17, R17);
RelationMake (StrR18, R18);
RelationMake (StrR19, R19);
RelationMake (StrR20, R20);
RelationMake (StrR21, R21);
RelationMake (StrR22, R22);
RelationMake (StrR23, R23);
RelationMake (StrR24, R24);
RelationMake (StrR25, R25);
RelationMake (StrR26, R26);
RelationMake (StrR27, R27);
RelationMake (StrR28, R28);
RelationMake (StrR29, R29);
RelationMake (StrR30, R30);
RelationMake (StrR1, R31);
RelationMake (StrR2, R32);
RelationMake (StrR3, R33);
RelationMake (StrR4, R34);
RelationMake (StrR5, R35);
RelationMake (StrR1, R36);
RelationMake (StrR2, R37);
RelationMake (StrR3, R38);
RelationMake (StrR4, R39);
RelationMake (StrR5, R40);

switch (NumberOfSubGraphs) { 
  case 10: 
    AddRNode (Cl, R1, FROM, L1, 1);
    AddRNode (C2, R1, TO, L1, 2);
    AddRNode (C1, R3, FROM, L1, 1);
    AddRNode (C4, R3, TO, L1, 3);
    AddRNode (C2, R5, FROM, L1, 2);
    AddRNode (C4, R5, TO, L1, 3);
    AddRNode (C5, R6, FROM, L1, 1);
    AddRNode (C6, R6, TO, L1, 2);
    AddRNode (C5, R8, FROM, L1, 1);
    AddRNode (C4, R8, TO, L1, 3);
    AddRNode (C6, R10, FROM, L1, 2);
    AddRNode (C4, R10, TO, L1, 3);
    AddRNode (C13, R21, FROM, L1, 1);
    AddRNode (C19, R21, TO, L1, 2);
    AddRNode (C13, R23, FROM, L1, 1);
    AddRNode (C14, R23, TO, L1, 3);
    AddRNode (C6, R25, FROM, L1, 2);
    AddRNode (C14, R25, TO, L1, 3);
    AddRNode (C18, R31, FROM, L1, 1);
    AddRNode (C19, R31, TO, L1, 2);
    AddRNode (C18, R33, FROM, L1, 1);
    AddRNode (C14, R33, TO, L1, 3);
    AddRNode (C19, R35, FROM, L1, 2);
    AddRNode (C14, R35, TO, L1, 3);
    AddRNode (C9, R11, FROM, L1, 1);
    AddRNode (C6, R11, TO, L1, 2);
    AddRNode (C9, R13, FROM, L1, 1);
    AddRNode (C10, R13, TO, L1, 3);
    AddRNode (C8, R15, FROM, L1, 2);
    AddRNode (C10, R15, TO, L1, 3);
    AddRNode (C12, R16, FROM, L1, 1);
    AddRNode (C11, R16, TO, L1, 2);
    AddRNode (C12, R18, FROM, L1, 1);
    AddRNode (C10, R18, TO, L1, 3);
    AddRNode (C11, R20, FROM, L1, 2);
    AddRNode (C10, R20, TO, L1, 3);
    AddRNode (C17, R26, FROM, L1, 1);
    AddRNode (C11, R26, TO, L1, 2);
    AddRNode (C17, R30, FROM, L1, 1);
    AddRNode (C18, R30, TO, L1, 2);
    AddRNode (C16, R30, FROM, L1, 2);
    AddRNode (C16, R30, TO, L1, 3);
    AddRNode (C22, R36, FROM, L1, 1);
    AddRNode (C21, R36, TO, L1, 2);
    AddRNode (C22, R38, FROM, L1, 1);
    AddRNode (C16, R38, TO, L1, 3);
    AddRNode (C21, R40, FROM, L1, 2);
    AddRNode (C16, R40, TO, L1, 3);
    AddRNode (C3, R2, FROM, L2, 2);
    AddRNode (C2, R2, TO, L2, 1);
    AddRNode (C3, R12, FROM, L2, 2);
    AddRNode (C8, R12, TO, L2, 2);
    AddRNode (C3, R4, FROM, L2, 3);
    AddRNode (C4, R4, TO, L2, 4);
    AddRNode (C3, R14, FROM, L2, 2);
    AddRNode (C10, R14, TO, L2, 5);
    AddRNode (C7, R9, FROM, L2, 6);
    AddRNode (C4, R9, TO, L2, 4);
    AddRNode (C7, R19, FROM, L2, 6);
    AddRNode (C10, R19, TO, L2, 5);
    AddRNode (C7, R7, FROM, L2, 6);
    AddRNode (C6, R7, TO, L2, 7);
    AddRNode (C7, R17, FROM, L2, 6);
    AddRNode (C11, R17, TO, L2, 8);
    AddRNode (C15, R22, FROM, L2, 2);
    AddRNode (C6, R22, TO, L2, 1);
    AddRNode (C15, R24, FROM, L2, 2);
    AddRNode (C14, R24, TO, L2, 4);
    AddRNode (C15, R27, FROM, L2, 2);
    AddRNode (C12, R27, TO, L2, 3);
    AddRNode (C15, R29, FROM, L2, 2);
    AddRNode (C16, R29, TO, L2, 5);
    AddRNode (C20, R32, FROM, L2, 6);
    AddRNode (C19, R32, TO, L2, 7);
    AddRNode (C20, R34, FROM, L2, 6);
    AddRNode (C14, R34, TO, L2, 4);
    AddRNode (C20, R37, FROM, L2, 6);
case 20:
    AddRNode (C21, R37, TO, L2, 8);
    AddRNode (C20, R39, FROM, L2, 6);
    AddRNode (C16, R39, TO, L2, 5);
    break;

    AddRNode (C1, R1, FROM, L1, 1);
    AddRNode (C2, R1, TO, L1, 2);
    AddRNode (C1, R3, FROM, L1, 1);
    AddRNode (C4, R3, TO, L1, 3);
    AddRNode (C5, R6, FROM, L1, 1);
    AddRNode (C6, R6, TO, L1, 2);
    AddRNode (C5, R8, FROM, L1, 1);
    AddRNode (C4, R8, TO, L1, 3);
    AddRNode (C13, R21, FROM, L1, 1);
    AddRNode (C6, R21, TO, L1, 2);
    AddRNode (C13, R23, FROM, L1, 1);
    AddRNode (C14, R23, TO, L1, 3);
    AddRNode (C18, R31, FROM, L1, 1);
    AddRNode (C19, R31, TO, L1, 2);
    AddRNode (C16, R33, FROM, L1, 1);
    AddRNode (C14, R33, TO, L1, 3);
    AddRNode (C9, R11, FROM, L1, 1);
    AddRNode (C8, R11, TO, L1, 2);
    AddRNode (C9, R13, FROM, L1, 1);
    AddRNode (C10, R13, TO, L1, 3);
    AddRNode (C12, R16, FROM, L1, 1);
    AddRNode (C11, R16, TO, L1, 2);
    AddRNode (C12, R18, FROM, L1, 1);
    AddRNode (C10, R18, TO, L1, 3);
    AddRNode (C17, R28, FROM, L1, 1);
    AddRNode (C11, R26, TO, L1, 2);
    AddRNode (C17, R28, FROM, L1, 1);
    AddRNode (C16, R28, TO, L1, 3);
    AddRNode (C22, R38, FROM, L1, 1);
    AddRNode (C21, R35, TO, L1, 2);
    AddRNode (C22, R38, FROM, L1, 1);
    AddRNode (C16, R33, TO, L1, 3);
    AddRNode (C3, R2, FROM, L2, 1);
    AddRNode (C2, R2, TO, L2, 2);
    AddRNode (C3, R4, FROM, L2, 1);
    AddRNode (C4, R4, TO, L2, 3);
    AddRNode (C7, R7, FROM, L2, 1);
    AddRNode (C6, R7, TO, L2, 2);
    AddRNode (C7, R9, FROM, L2, 1);
    AddRNode (C4, R9, TO, L2, 3);
    AddRNode (C15, R22, FROM, L2, 1);
    AddRNode (C6, R22, TO, L2, 2);
    AddRNode (C15, R24, FROM, L2, 1);
    AddRNode (C14, R24, TO, L2, 3);
    AddRNode (C20, R32, FROM, L2, 1);
    AddRNode (C19, R32, TO, L2, 2);
    AddRNode (C20, R34, FROM, L2, 1);
    AddRNode (C14, R34, TO, L2, 3);
    AddRNode (C3, R12, FROM, L2, 1);
    AddRNode (C8, R12, TO, L2, 2);
    AddRNode (C3, R14, FROM, L2, 1);
    AddRNode (C10, R14, TO, L2, 3);
    AddRNode (C7, R17, FROM, L2, 1);
    AddRNode (C11, R17, TO, L2, 2);
    AddRNode (C7, R19, FROM, L2, 1);
    AddRNode (C10, R19, TO, L2, 3);
    AddRNode (C15, R27, FROM, L2, 1);
    AddRNode (C11, R27, TO, L2, 2);
    AddRNode (C15, R29, FROM, L2, 1);
    AddRNode (C16, R29, TO, L2, 3);
    AddRNode (C20, R37, FROM, L2, 1);
    AddRNode (C21, R37, TO, L2, 2);
    AddRNode (C20, R39, FROM, L2, 1);
    AddRNode (C16, R39, TO, L2, 3);
    AddRNode (C2, R5, FROM, L3, 2);
    AddRNode (C4, R5, TO, L3, 1);
    AddRNode (C6, R10, FROM, L3, 3);
    AddRNode (C4, R10, TO, L3, 3);
    AddRNode (C8, R15, FROM, L3, 2);
    AddRNode (C10, R35, TO, L3, 1);
    AddRNode (C11, R20, FROM, L3, 1);
    AddRNode (C10, R20, TO, L3, 3);
    AddRNode (C6, R25, FROM, L3, 2);
    AddRNode (C14, R25, TO, L3, 1);
    AddRNode (C19, R35, FROM, L3, 1);
    AddRNode (C14, R35, TO, L3, 3);
    AddRNode (C13, R30, FROM, L3, 2);
    AddRNode (C16, R30, TO, L3, 1);
    AddRNode (C21, R40, FROM, L3, 1);
    AddRNode (C16, R40, TO, L3, 3);
    break;

case 40:
    AddRNode (C1, R1, FROM, L1, 1);
    AddRNode (C2, R1, TO, L1, 2);
    AddRNode (C3, R2, FROM, L2, 1);
AddRNode (C2, R2, TO, L2, 2);
AddRNode (C1, R3, FROM, L3, 1);
AddRNode (C4, R3, TO, L3, 2);
AddRNode (C3, R4, FROM, L4, 1);
AddRNode (C4, R4, TO, L4, 2);
AddRNode (C2, R5, FROM, L5, 1);
AddRNode (C4, R5, TO, L5, 1);
AddRNode (C5, R6, FROM, L1, 1);
AddRNode (C6, R6, TO, L1, 2);
AddRNode (C7, R7, FROM, L2, 1);
AddRNode (C5, R8, FROM, L3, 1);
AddRNode (C4, R8, TO, L3, 2);
AddRNode (C7, R9, FROM, L4, 1);
AddRNode (C4, R9, TO, L4, 2);
AddRNode (C6, R10, FROM, L5, 1);
AddRNode (C4, R10, TO, L5, 2);
AddRNode (C9, R11, FROM, L1, 1);
AddRNode (C8, R11, TO, L1, 2);
AddRNode (C3, R12, FROM, L2, 1);
AddRNode (C6, R12, TO, L2, 2);
AddRNode (C9, R13, FROM, L3, 1);
AddRNode (C10, R13, TO, L3, 2);
AddRNode (C1, R14, FROM, L4, 1);
AddRNode (C10, R14, TO, L4, 2);
AddRNode (C8, R15, FROM, L5, 1);
AddRNode (C10, R15, TO, L5, 1);
AddRNode (C12, R16, FROM, L1, 1);
AddRNode (C11, R16, TO, L1, 2);
AddRNode (C7, R17, FROM, L2, 1);
AddRNode (C11, R17, TO, L2, 2);
AddRNode (C12, R18, FROM, L3, 1);
AddRNode (C10, R18, TO, L3, 2);
AddRNode (C7, R19, FROM, L4, 1);
AddRNode (C10, R19, TO, L4, 2);
AddRNode (C11, R20, FROM, L5, 1);
AddRNode (C19, R20, TO, L5, 1);
AddRNode (C13, R21, FROM, L1, 1);
AddRNode (C6, R21, TO, L1, 2);
AddRNode (C15, R22, FROM, L2, 1);
AddRNode (C6, R22, TO, L2, 2);
AddRNode (C13, R23, FROM, L3, 1);
AddRNode (C14, R23, TO, L3, 2);
AddRNode (C15, R24, FROM, L4, 1);
AddRNode (C14, R24, TO, L4, 2);
AddRNode (C6, R25, FROM, L5, 1);
AddRNode (C14, R25, TO, L5, 1);
AddRNode (C17, R26, FROM, L1, 1);
AddRNode (C11, R26, TO, L1, 2);
AddRNode (C13, R27, FROM, L2, 1);
AddRNode (C11, R27, TO, L2, 2);
AddRNode (C17, R28, FROM, L3, 1);
AddRNode (C16, R28, TO, L3, 2);
AddRNode (C15, R29, FROM, L4, 1);
AddRNode (C16, R29, TO, L4, 2);
AddRNode (C11, R30, FROM, L5, 1);
AddRNode (C16, R30, TO, L5, 1);
AddRNode (C18, R31, FROM, L1, 1);
AddRNode (C19, R31, TO, L1, 2);
AddRNode (C20, R32, FROM, L2, 1);
AddRNode (C19, R32, TO, L2, 2);
AddRNode (C18, R33, FROM, L3, 1);
AddRNode (C14, R33, TO, L3, 2);
AddRNode (C20, R34, FROM, L4, 1);
AddRNode (C14, R34, TO, L4, 2);
AddRNode (C19, R35, FROM, L5, 1);
AddRNode (C14, R35, TO, L5, 1);
AddRNode (C22, R36, FROM, L1, 1);
AddRNode (C21, R36, TO, L1, 2);
AddRNode (C20, R37, FROM, L2, 1);
AddRNode (C21, R37, TO, L2, 2);
AddRNode (C22, R38, FROM, L3, 1);
AddRNode (C16, R38, TO, L3, 2);
AddRNode (C20, R39, FROM, L4, 1);
AddRNode (C16, R39, TO, L4, 2);
AddRNode (C21, R40, FROM, L5, 1);
AddRNode (C16, R40, TO, L5, 1);
break;
break;

Complex = typealloc (ConceptListElt);
Complex_Cxt = typealloc (Context);
Complex_SimpleGraphPtr = typealloc (ConceptListElt);
Complex->PointedAt = Complex_Cxt;
Complex->NextInList = 0;
Complex->Used = 'N';
Complex->Ident = NOTYETDEF;
Complex_Cxt->Depth = 0;
Complex_Cxt->Dominating = 0;
Complex_Cxt->DominatedBy = 0;
Complex_Cxt->Processed = NOTYETDEF;
Complex_SimpleGraphPtr->ListOfMembers = Complex_SimpleGraphPtr;
Complex_SimpleGraphPtr->PointerType = Cl;
Complex_SimpleGraphPtr->Used = 'N';
Complex_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_SimpleGraphPtr->NextInList = 0;

CList = typealloc (ContentListElt);
JList = typealloc (JoinListElt);
RList = typealloc (RestrictionListElt);
CList->PointedAt = D.
CList->NextInList = D.
strncpy (RList->AliasName, '\0', 25);
RList->NextInList = 0;
strncpy (JList->TableName, '\0', 25);
JList->NextInList = 0;
CF = FIRST;
JF = FIRST;
RF = FIRST;
CFlag = &CF;
JFlag = &JF;
RFlag = &RF;
strncpy (InitTab, '\0', 25);
strncpy (InitTab. IT, 2);
Instruction = (char *)malloc (5000);
SecondInstruction = (char *)malloc (5000);

/***********************************************************/
/* Start timing of operation and form lists. */
/***********************************************************/

start = times (&tmsstart);
Complex_Query_Maker (Complex, CList, JList, RList, CFlag, JFlag, RFlag, InitTab, TLattice);
for (Scrub = Complex; Scrub != 0; Scrub = Scrub->NextInList){
   ScrubaTarget = Scrub->PointedAt;
    ScrubaTarget->Processed = NOTYETDEF;
    for (ScrubMemb = ScrubaTarget->ListOfMembers; ScrubMemb != 0; ScrubMemb = ScrubMemb->NextInList){
        Backtrack_Over_Flags (ScrubMemb->PointedAt);
    }
}

/***********************************************************/
/* Now form a view. */
/***********************************************************/

strcpy (LastView, MakeAlias (LastView, 'v'));
Instruction = FormSQLCall (CList, JList, RList, LastView, TRUE, TLattice);

/***********************************************************/
/* Now form a selection upon that view, and stop timing. */
/***********************************************************/

SecondInstruction = FormSQLCall (CList, JList, RList, LastView, FALSE, TLattice);
end = times (&tmsend);

printf ("View Formation Instruction: %s\n", Instruction);
printf ("Select Instruction : %s\n", SecondInstruction);
printf ("\n\nTime taken to compute Phi-Delta instructions for Dyad Size %d, SubGraphs %d:\n\n",
    DyadInGraph, NumberOfSubGraphs);
pr_times (end-start, &tmsstart, &tmsend);
}

void main ()
{
    Phi_Delta_Operate (10, 10);
    Phi_Delta_Operate (20, 10);
    Phi_Delta_Operate (20, 20);
    Phi_Delta_Operate (40, 10);
    Phi_Delta_Operate (40, 20);
    Phi_Delta_Operate (40, 40);
    Phi_Delta_Operate (80, 10);
    Phi_Delta_Operate (80, 20);
    Phi_Delta_Operate (80, 40);
    Phi_Delta_Operate (80, 80);
    exit (0);
}
Appendix G
Building a Domain:
Familial Relationships

This appendix shows how a typical application domain is built up from scratch. User input is in bold, whilst references to this appendix in the main thesis (Section 5.4) are in italic.

cgrass

$\textbf{Input, Please}

(0a) \texttt{DISPLAY LATTICE END}

$\textbf{Type Hierarchy Details}$

\begin{itemize}
  \item UNIVERSAL $>$ STRING
  \item UNIVERSAL $>$ NUMERIC
  \item UNIVERSAL $>$ INDIVIDUAL
\end{itemize}

$\textbf{Press RETURN to look at the relation hierarchy...}$

$\textbf{Relation Hierarchy Details}$

\begin{itemize}
  \item UNIVERSAL $>$ ARG1
  \item UNIVERSAL $>$ ARG2
  \item UNIVERSAL $>$ RSLT
\end{itemize}

Time taken (seconds) to perform last macro:

\begin{itemize}
  \item real: 0.5600000
  \item user: 0.0000000
  \item sys: 0.0000000
  \item child user: 0.0000000
  \item child sys: 0.0000000
\end{itemize}

$\textbf{What Next?}$

(0b) \texttt{DISPLAY CATALOGUE END}

$\textbf{Message from Display Subsystem...}$

$\textbf{The canonical model graph is empty at present}$

Time taken (seconds) to perform last macro:

\begin{itemize}
  \item real: 0.0000000
  \item user: 0.0000000
  \item sys: 0.0000000
  \item child user: 0.0000000
  \item child sys: 0.0000000
\end{itemize}

$\textbf{What Next?}$

(0c) \texttt{DISPLAY GRAPH END}

$\textbf{Message from Display Subsystem...}$

$\textbf{The first-order graph is empty at present}$

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(1) ASSERT LATTICE STRING >> FAMILY_NAME END

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(1a) DISPLAY LATTICE END

(2) ASSERT LATTICE STRING >> COMPANY_NAME END

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(2a) DISPLAY LATTICE END
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$$ What Next? $$

(3) ASSERT LATTICE STRING >> PERSON_NAME END

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$$ What Next? $$

(3a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME

$$ Press RETURN to look at the relation hierarchy... $$

$$ Relation Hierarchy Details $$

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:

real: 0.170000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$$ What Next? $$

(4) ASSERT LATTICE INDIVIDUAL >> PERSON END

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$$ What Next? $$

(4a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON

$$ Press RETURN to look at the relation hierarchy... $$

$$ Relation Hierarchy Details $$

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:
real: 0.1300000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$$\text{What Next?}$$

(5) \texttt{ASSERT LATTICE INDIVIDUAL >> ACT END}

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$\text{What Next?}$$

(5a) \texttt{DISPLAY LATTICE END}

$$\text{Type Hierarchy Details}$$

\texttt{UNIVERSAL >> STRING}
\texttt{UNIVERSAL >> NUMERIC}
\texttt{UNIVERSAL >> INDIVIDUAL}
\texttt{STRING >> \text{FAMILY\_NAME}}
\texttt{STRING >> \text{COMPANY\_NAME}}
\texttt{INDIVIDUAL >> \text{PERSON}}
\texttt{INDIVIDUAL >> \text{ACT}}

$$\text{Press RETURN to look at the relation hierarchy...}$$

$$\text{Relation Hierarchy Details}$$

\texttt{UNIVERSAL >> ARG1}
\texttt{UNIVERSAL >> ARG2}
\texttt{UNIVERSAL >> RSLT}

Time taken (seconds) to perform last macro:
real: 0.1200000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$\text{What Next?}$$

(6) \texttt{ASSERT LATTICE INDIVIDUAL >> MALENESS END}

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$\text{What Next?}$$

(6a) \texttt{DISPLAY LATTICE END}

$$\text{Type Hierarchy Details}$$

\texttt{UNIVERSAL >> STRING}
\texttt{UNIVERSAL >> NUMERIC}
\texttt{UNIVERSAL >> INDIVIDUAL}
\texttt{STRING >> \text{FAMILY\_NAME}}
\texttt{STRING >> \text{COMPANY\_NAME}}
\texttt{STRING >> \text{PERSON\_NAME}}
\texttt{INDIVIDUAL >> \text{PERSON}}
\texttt{INDIVIDUAL >> \text{ACT}}
\texttt{INDIVIDUAL >> MALENESS}

$$\text{Press RETURN to look at the relation hierarchy...}$$

$$\text{Relation Hierarchy Details}$$
UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:
real:  0.1200000
user:  0.0000000
sys:  0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ \text{What Next?} $$

(7) ASSERT LATTICE INDIVIDUAL >> FEMALENESS END

Time taken (seconds) to perform last macro:
real:  0.0000000
user:  0.0000000
sys:  0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ \text{What Next?} $$

(7a) DISPLAY LATTICE END

$$ \text{Type Hierarchy Details} $$

$$ \text{UNIVERSAL >> STRING} $$
$$ \text{UNIVERSAL >> NUMERIC} $$
$$ \text{UNIVERSAL >> INDIVIDUAL} $$
$$ \text{STRING >> FAMILY_NAME} $$
$$ \text{STRING >> COMPANY_NAME} $$
$$ \text{STRING >> PERSON_NAME} $$
$$ \text{INDIVIDUAL >> PERSON} $$
$$ \text{INDIVIDUAL >> ACT} $$
$$ \text{INDIVIDUAL >> MALENESS} $$
$$ \text{INDIVIDUAL >> FEMALENESS} $$

$$ \text{Press RETURN to look at the relation hierarchy...} $$

$$ \text{Relation Hierarchy Details} $$

$$ \text{UNIVERSAL > ARG1} $$
$$ \text{UNIVERSAL > ARG2} $$
$$ \text{UNIVERSAL > RSLT} $$

Time taken (seconds) to perform last macro:
real:  0.1000000
user:  0.0100000
sys:  0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ \text{What Next?} $$

(8) ASSERT LATTICE INDIVIDUAL >> FAMILY END

Time taken (seconds) to perform last macro:
real:  0.0100000
user:  0.0100000
sys:  0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ \text{What Next?} $$

(8a) DISPLAY LATTICE END

$$ \text{Type Hierarchy Details} $$

$$ \text{UNIVERSAL >> STRING} $$
$$ \text{UNIVERSAL >> NUMERIC} $$
$$ \text{UNIVERSAL >> INDIVIDUAL} $$
$$ \text{STRING >> FAMILY_NAME} $$
$$ \text{STRING >> COMPANY_NAME} $$
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY

$§ Press RETURN to look at the relation hierarchy...

$§ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:

real: 0.1200000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$§ What Next?

(9) ASSERT LATTICE INDIVIDUAL >> STATE END

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$§ What Next?

(9a) DISPLAY LATTICE END

$§ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE

$§ Press RETURN to look at the relation hierarchy...

$§ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:

real: 0.1600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$§ What Next?

(10) ASSERT LATTICE PERSON >> MAN

$§ Message From Assertion Subsystem...

$§ The type MAN is previously unknown
$§ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.9800000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(10a) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL >> ARG1
UNIVERSAL >> ARG2
UNIVERSAL >> RSLT

Time taken (seconds) to perform last macro:

real: 0.2400000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(II) ASSERT LATTICE PERSON >> WOMAN END

$\$ Message From Assertion Subsystem...
$\$ The type WOMAN is previously unknown
$\$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.2400000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(IIa) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL >> ARG1
UNIVERSAL >> ARG2
UNIVERSAL >> RSLT
Time taken (seconds) to perform last macro:

real: 0.130000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?

(12) ASSERT LATTICE STATE >> MARRIAGE END

$ Message From Assertion Subsystem...
$ The type MARRIAGE is previously unknown
$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.230000
user: 0.010000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?

(12a) DISPLAY LATTICE END

$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE

$ Press RETURN to look at the relation hierarchy...

$ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT

Time taken (seconds) to perform last macro:

real: 0.120000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?

(13) ASSERT LATTICE ACT >> BIRTH END

$ Message From Assertion Subsystem...
$ The type BIRTH is previously unknown
$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.250000
user: 0.012000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?

(13a) DISPLAY LATTICE END

$ Type Hierarchy Details
SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

 UNIVERSAL > ARG1
 UNIVERSAL > ARG2
 UNIVERSAL > RSLT

 Time taken (seconds) to perform last macro:

 real: 0.1200000
 user: 0.0000000
 sys: 0.0000000
 child user: 0.0000000
 child sys: 0.0000000

 SS What Next?

 (14) ASSERT LATTICE (UNIVERSAL) > (NAME) END

 Time taken (seconds) to perform last macro:

 real: 0.0100000
 user: 0.0100000
 sys: 0.0000000
 child user: 0.0000000
 child sys: 0.0000000

 SS What Next?

 (14a) DISPLAY LATTICE END

 SS Type Hierarchy Details

 UNIVERSAL > STRING
 UNIVERSAL > NUMERIC
 UNIVERSAL > INDIVIDUAL
 STRING > FAMILY_NAME
 STRING > COMPANY_NAME
 STRING > PERSON_NAME
 INDIVIDUAL > PERSON
 INDIVIDUAL > ACT
 INDIVIDUAL > MALENESS
 INDIVIDUAL > FEMALENESS
 INDIVIDUAL > FAMILY
 INDIVIDUAL > STATE
 PERSON > MAN
 PERSON > WOMAN
 STATE > MARRIAGE
 ACT > BIRTH

 SS Press RETURN to look at the relation hierarchy...

 SS Relation Hierarchy Details

 UNIVERSAL > ARG1
 UNIVERSAL > ARG2
 UNIVERSAL > RSLT
 UNIVERSAL > NAME

 Time taken (seconds) to perform last macro:

 real: 0.0800000
 user: 0.0000000
 sys: 0.0000000
 child user: 0.0000000
 child sys: 0.0000000
child sys: 0.0000000

$\$\$ What Next?

(15) ASSERT LATTICE (UNIVERSAL) > (FRIEND) END

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(15a) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > NAME
UNIVERSAL > FRIEND

Time taken (seconds) to perform last macro:
real: 0.0900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(16) ASSERT LATTICE (UNIVERSAL) > (GENDER) END

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(16a) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
$\text{INDIVIDUAL} \gg \text{FEMALENESS}
\text{INDIVIDUAL} \gg \text{FAMILY}
\text{INDIVIDUAL} \gg \text{STATE}
\text{PERSON} \gg \text{MAN}
\text{PERSON} \gg \text{WOMAN}
\text{STATE} \gg \text{MARRIAGE}
\text{ACT} \gg \text{BIRTH}

$\text{SS Press RETURN to look at the relation hierarchy...}$

$\text{SS Relation Hierarchy Details}$

$\text{UNIVERSAL} \gg \text{ARG1}$
$\text{UNIVERSAL} \gg \text{ARG2}$
$\text{UNIVERSAL} \gg \text{RSLT}$
$\text{UNIVERSAL} \gg \text{FRIEND}$
$\text{UNIVERSAL} \gg \text{NAME}$
$\text{UNIVERSAL} \gg \text{GENDER}$

Time taken (seconds) to perform last macro:

real: 0.1200000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{SS What Next?}$

(17) \text{ASSERT LATTICE (UNIVERSAL) > (LEGAL_PARTNER) END}

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{SS What Next?}$

(17a) \text{DISPLAY LATTICE END}

$\text{SS Type Hierarchy Details}$

$\text{UNIVERSAL} \gg \text{STRING}$
$\text{UNIVERSAL} \gg \text{NUMERIC}$
$\text{UNIVERSAL} \gg \text{INDIVIDUAL}$
$\text{STRING} \gg \text{FAMILY_NAME}$
$\text{STRING} \gg \text{COMPANY_NAME}$
$\text{STRING} \gg \text{PERSON_NAME}$
$\text{INDIVIDUAL} \gg \text{PERSON}$
$\text{INDIVIDUAL} \gg \text{ACT}$
$\text{INDIVIDUAL} \gg \text{MALENESS}$
$\text{INDIVIDUAL} \gg \text{FEMALENESS}$
$\text{INDIVIDUAL} \gg \text{FAMILY}$
$\text{INDIVIDUAL} \gg \text{STATE}$
$\text{PERSON} \gg \text{MAN}$
$\text{PERSON} \gg \text{WOMAN}$
$\text{STATE} \gg \text{MARRIAGE}$
$\text{ACT} \gg \text{BIRTH}$

$\text{SS Press RETURN to look at the relation hierarchy...}$

$\text{SS Relation Hierarchy Details}$

$\text{UNIVERSAL} \gg \text{ARG1}$
$\text{UNIVERSAL} \gg \text{ARG2}$
$\text{UNIVERSAL} \gg \text{RSLT}$
$\text{UNIVERSAL} \gg \text{NAME}$
$\text{UNIVERSAL} \gg \text{GENDER}$
$\text{UNIVERSAL} \gg \text{LEGAL_PARTNER}$

Time taken (seconds) to perform last macro:

real: 0.0900000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{SS What Next?}$
(18) ASSERT LATTICE (UNIVERSAL) > (MEMB) END

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(18a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy... $$

$$ Relation Hierarchy Details $$

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > FRIEND
UNIVERSAL > NAME
UNIVERSAL > MEMB

Time taken (seconds) to perform last macro:

real: 0.1000000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(19) ASSERT LATTICE (UNIVERSAL) > (RELATIVE) END

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(19a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$S Press RETURN to look at the relation hierarchy...

$S Relation Hierarchy Details

UNIVERSAL >> ARG1
UNIVERSAL >> ARG2
UNIVERSAL >> RSLT
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> MEMB
UNIVERSAL >> NAME
UNIVERSAL >> FRIEND
UNIVERSAL >> GENDER
UNIVERSAL >> RELATIVE

Time taken (seconds) to perform last macro:
real: 0.1000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$S What Next?

(20) ASSERT LATTICE (UNIVERSAL) > (AGNT) END

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$S What Next?

(20a) DISPLAY LATTICE END

$S Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$S Press RETURN to look at the relation hierarchy...

$S Relation Hierarchy Details

UNIVERSAL >> ARG1
UNIVERSAL >> ARG2
UNIVERSAL >> RSLT
UNIVERSAL >> MEMB
UNIVERSAL >> GENDER
UNIVERSAL >> FRIEND
UNIVERSAL >> NAME
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT

Time taken (seconds) to perform last macro:
real: 0.1000000
user: 0.0100000
sys: 0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ What Next?

(21) ASSERT LATTICE (UNIVERSAL) > (EXPR) END

Time taken (seconds) to perform last macro:

real:   0.0000000
user:   0.0000000
sys:    0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ What Next?

(21a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > AGNT
UNIVERSAL > NAME
UNIVERSAL > FRIEND
UNIVERSAL > MEMB
UNIVERSAL > RELATIVE
UNIVERSAL > EXPR

Time taken (seconds) to perform last macro:

real:   0.1000000
user:   0.0000000
sys:    0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ What Next?

(22) ASSERT LATTICE (UNIVERSAL) > (ARG1) END

$$ Message From Assertion Subsystem...

$$ Logical arc already exists, directly or indirectly
$$ Therefore, the arc will not be scored

Time taken (seconds) to perform last macro:

real:   0.0000000
user:   0.0000000
sys:    0.0000000
child user:  0.0000000
child sys:  0.0000000

$$ What Next?

(23) ASSERT LATTICE (UNIVERSAL) > (ARG2) END
Message From Assertion Subsystem...
Logical arc already exists, directly or indirectly.
Therefore, the arc will not be stored.

Time taken (seconds) to perform last macro:
- real: 0.0000000
- user: 0.0000000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

What Next?
(24) ASSERT LATTICE (UNIVERSAL) > (RESULT) END

Message From Assertion Subsystem...
Logical arc already exists, directly or indirectly.
Therefore, the arc will not be stored.

Time taken (seconds) to perform last macro:
- real: 0.0000000
- user: 0.0000000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

What Next?
(25) ASSERT LATTICE (UNIVERSAL) > (RESULT) END

Time taken (seconds) to perform last macro:
- real: 0.0100000
- user: 0.0100000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

What Next?
(25a) DISPLAY LATTICE END

Type Hierarchy Details
- UNIVERSAL >> STRING
- UNIVERSAL >> NUMERIC
- UNIVERSAL >> INDIVIDUAL
- STRING >> FAMILY_NAME
- STRING >> COMPANY_NAME
- STRING >> PERSON_NAME
- INDIVIDUAL >> PERSON
- INDIVIDUAL >> ACT
- INDIVIDUAL >> MALENESS
- INDIVIDUAL >> FEMALENNESS
- INDIVIDUAL >> FAMILY
- INDIVIDUAL >> STATE
- PERSON >> MAN
- PERSON >> WOMAN
- STATE >> MARRIAGE
- ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details
- UNIVERSAL > ARG1
- UNIVERSAL > ARG2
- UNIVERSAL > RELT
- UNIVERSAL > EXPR
- UNIVERSAL > MEMB
- UNIVERSAL > FRIEND
- UNIVERSAL > RELATIVE
- UNIVERSAL > AGNT
- UNIVERSAL > LEGAL_PARTNER
- UNIVERSAL > NAME
- UNIVERSAL > GENDER
- UNIVERSAL > RESULT
Time taken (seconds) to perform last macro:

real: 0.1000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$\$ What Next?

(26) ASSERT LATTICE (RELATIVE) > (BLOOD_RELATIVE) END

$\$ Message From Assertion Subsystem...
$\$ The relation BLOOD_RELATIVE is previously unknown
$\$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.1800000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(26a) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALNESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > MEMB
UNIVERSAL > AGNT
UNIVERSAL >> NAME
UNIVERSAL >> RESULT
UNIVERSAL >> GENDER
UNIVERSAL >> RELATIVE
UNIVERSAL >> FRIEND
UNIVERSAL >> EXPR
UNIVERSAL >> LEGAL_PARTNER
RELATIVE > BLOOD_RELATIVE

Time taken (seconds) to perform last macro:

real: 0.3400000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(27) ASSERT LATTICE (RELATIVE) > (IN_LAW_RELATIVE) END

$\$ Message From Assertion Subsystem...
$\$ The relation IN_LAW_RELATIVE is previously unknown
$\$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.1500000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(27a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy... $$

$$ Relation Hierarchy Details $$

UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > AGNT
UNIVERSAL > GENDER
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > EXPR
UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > MEMBER
UNIVERSAL > RELATIVE
RELATIVE > BLOOD_RELATIVE
RELATIVE > IN_LAW_RELATIVE

Time taken (seconds) to perform last macro:

real: 0.0900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(28) ASSERT LATTICE (BLOOD_RELATIVE) > (CHILD) END

$$ Message From Assertion Subsystem... $$

$$ The relation BLOOD_RELATIVE has no definition $$
$$ Do you wish to define it (Y/N)? N $$

$$ Message From Assertion Subsystem... $$

$$ The relation CHILD is previously unknown $$
$$ Do you wish to define it (Y/N)? N $$

Time taken (seconds) to perform last macro:

real: 0.4500000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next? $$

(28a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME

(27a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$

Press RETURN to look at the relation hierarchy...

$\$

Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > EXPRT
UNIVERSAL > GENDER
UNIVERSAL > REL
UNIVERSAL > ARG2
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
BLOOD_RELATIVE > CHILD

Time taken (seconds) to perform last macro:

real: 0.0900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$

What Next?

(29) ASSERT LATTICE (IN_LAW_RELATIVE) \(\rightarrow\) (SPOUSE) END

$\$

Message From Assertion Subsystem...

The relation IN_LAW_RELATIVE has no definition
Do you wish to define it (Y/N)? N

$\$

Message From Assertion Subsystem...

The relation SPOUSE is previously unknown
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.4500000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$

What Next?

(29a) DISPLAY LATTICE END

$\$

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$

Press RETURN to look at the relation hierarchy...
$$ Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > GENDER
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > NAME
UNIVERSAL > MEMBER
UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RESULT
RELATIVE > BLOOD_RELATIVE
RELATIVE > IN_LAW_RELATIVE
BLOOD_RELATIVE > CHILD
IN_LAW_RELATIVE > SPOUSE

Time taken (seconds) to perform last macro:
real: 0.1300000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(30a) DISPLAY LATTICE END

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RESULT
UNIVERSAL > EXPRESSION
UNIVERSAL > MEMBER
UNIVERSAL > NAME
UNIVERSAL > GENDER
UNIVERSAL > ARG2
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

Time taken (seconds) to perform last macro:

real: 0.1100000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(31) ASSERT LATTICE (IN_LAW_RELATIVE) > (PARENT_IN_LAW) END

$\$ Message From Assertion Subsystem...
$\$ The relation IN_LAW_RELATIVE has no definition
$\$ Do you wish to define it (Y/N)? N

$\$ Message From Assertion Subsystem...
$\$ The relation PARENT_IN_LAW is previously unknown
$\$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.4700000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(31a) DISPLAY LATTICE END

$\$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> EXPR
UNIVERSAL >> GENDER
UNIVERSAL >> RSLT
UNIVERSAL >> MEMB
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> ARG2
UNIVERSAL >> NAME
UNIVERSAL >> ARG1
UNIVERSAL >> RESULT
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SIBLING_IN_LAW
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
BLOOD_RELATIVE >> CHILD

Time taken (seconds) to perform last macro:
real: 1.0000000
user: 0.0100000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

What Next?

(32) ASSERT LATTICE (CHILD) > (BLOOD_SIBLING) END

Message From Assertion Subsystem...

The relation CHILD has no definition
Do you wish to define it (Y/N)? N

Message From Assertion Subsystem...

The relation BLOOD_SIBLING is previously unknown
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.4600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(32a) DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > GENDER
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > NAME
UNIVERSAL > RESULT
UNIVERSAL > ARG1
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > EXPR
UNIVERSAL > BLOOD_RELATIVE
RELATIVE > BLOOD_RELATIVE
RELATIVE > IN_LAW_RELATIVE
BLOOD_RELATIVE >> CHILD
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> SIBLING_IN_LAW
IN_LAW_RELATIVE >> PARENT_IN_LAW

Press RETURN to continue...

Relation Hierarchy Details

CHILD > BLOOD_SIBLING

Time taken (seconds) to perform last macro:
real: 0.2900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000
What Next?

 ASSERT LATTICE (CHILD) > (SIBLING_CHILD) END

Message From Assertion Subsystem...
The relation CHILD has no definition
Do you wish to define it (Y/N)? N

Message From Assertion Subsystem...
The relation SIBLING_CHILD is previously unknown
Do you wish to define it (Y/N)? N
Time taken (seconds) to perform last macro:

  real: 0.4600000
  user: 0.0100000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

What Next?

DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENES
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> ARG1
UNIVERSAL >> RSLT
UNIVERSAL >> ARG2
UNIVERSAL >> EXPR
UNIVERSAL >> RESULT
UNIVERSAL >> NAME
UNIVERSAL >> GENDER
UNIVERSAL >> MEMB
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SIBLING_IN_LAW
BLOOD_RELATIVE >> CHILD

Press RETURN to continue...

Relation Hierarchy Details

CHILD >> BLOOD_SIBLING
CHILD >> SIBLING_CHILD

Time taken (seconds) to perform last macro:

  real: 0.2600000
  user: 0.0100000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

What Next?
(34) ASSERT LATTICE (CHILD) > (BLOOD_PARENT) END

$$ Message From Assertion Subsystem...
$$ The relation CHILD has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation BLOOD_PARENT is previously unknown
$$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.4600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(35) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > ARG1
UNIVERSAL > EXPR
UNIVERSAL > NAME
UNIVERSAL > MEMB
UNIVERSAL > GENDER
UNIVERSAL > ARG2
UNIVERSAL > LSLT
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > RESULT
RELATIVE > BLOOD_RELATIVE
RELATIVE > IN_LAW_RELATIVE
BLOOD_RELATIVE > CHILD
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > SIBLING_IN_LAW

$$ Press RETURN to continue...

$$ Relation Hierarchy Details

CHILD > SIBLING_CHILD
CHILD > BLOOD_SIBLING
CHILD > BLOOD_PARENT

Time taken (seconds) to perform last macro:

real: 0.2700000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(35) ASSERT LATTICE (SIBLING_IN_LAW) > (BROTHER_IN_LAW) END

$$ Message From Assertion Subsystem...
The relation SIBLING_IN_LAW has no definition
Do you wish to define it (Y/N)? N

Message From Assertion Subsystem...
The relation BROTHER_IN_LAW is previously unknown
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.5100000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> EXPR
UNIVERSAL >> GENDER
UNIVERSAL >> RSLT
UNIVERSAL >> RESULT
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> MEMB
UNIVERSAL >> NAME
UNIVERSAL >> ARG!
UNIVERSAL >> ARG2
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SIBLING_IN_LAW
IN_LAW_RELATIVE >> SPOUSE
BLOOD_RELATIVE >> CHILD

Press RETURN to continue...

Relation Hierarchy Details

SIBLING_IN_LAW >> BROTHER_IN_LAW
CHILD >> BLOOD_PARENT
CHILD >> BLOOD_SIBLING
CHILD >> SIBLING_CHILD

Time taken (seconds) to perform last macro:
real: 0.2400000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

ASSERT LATTICE (SIBLING_IN_LAW) >> (SISTER_IN_LAW) END

Message From Assertion Subsystem...
The relation SIBLING_IN_LAW has no definition
Do you wish to define it (Y/N)? N
SS Message From Assertion Subsystem...
SS The relation SISTER_IN_LAW is previously unknown
SS Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3900000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?
(36a) DISPLAY LATTICE END

SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> GENDER
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> NAME
UNIVERSAL >> ARG2
UNIVERSAL >> ARG1
UNIVERSAL >> RESULT
UNIVERSAL >> RSLT
UNIVERSAL >> EXPR
UNIVERSAL >> MEMB
RELATIVE >> BLOOD_RELATIVE
RELATIVE >> IN_LAW_RELATIVE
BLOOD_RELATIVE >> CHILD
IN_LAW_RELATIVE >> SIBLING_IN_LAW
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SPOUSE

SS Press RETURN to continue...

SS Relation Hierarchy Details

CHILD >> SIBLINGCHILD
CHILD >> BLOOD_SIBLING
CHILD >> BLOOD_PARENT
SIBLING_IN_LAW >> BROTHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW

Time taken (seconds) to perform last macro:

real: 0.2500000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?
(37) ASSERT LATTICE (PARENT_IN_LAW) >> (FATHER_IN_LAW) END

SS Message From Assertion Subsystem...
SS The relation PARENT_IN_LAW has no definition
SS Do you wish to define it (Y/N)? N
SS Message From Assertion Subsystem...
SS The relation FATHER_IN-LAW is previously unknown
SS Do you wish to define it [Y/N]? N
Time taken (seconds) to perform last macro:

real: 0.4000000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(37a) DISPLAY LATTICE END

SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENES
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG1
UNIVERSAL > RSLT
UNIVERSAL > MEMB
UNIVERSAL > EXPR
UNIVERSAL > ARG2
UNIVERSAL > NAME
UNIVERSAL > GENDER
UNIVERSAL > RESULT
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SIBLING_IN_LAW
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN-LAW
BLOOD_RELATIVE > CHILD

SS Press RETURN to continue...

SS Relation Hierarchy Details

SIBLING_IN_LAW > SISTER_IN-LAW
SIBLING_IN_LAW > BROTHER_IN-LAW
PARENT_IN_LAW > FATHER_IN-LAW
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD

Time taken (seconds) to perform last macro:

real: 0.2500000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(38) ASSERT LATTICE (PARENT_IN_LAW) > (MOTHER_IN_LAW) END

SS Message From Assertion Subsystem...
SS The relation PARENT_IN_LAW has no definition
SS Do you wish to define it [Y/N]? N
SS Message From Assertion Subsystem...
SS The relation MOTHER_IN_LAW is previously unknown
SS Do you wish to define it (Y/N)? N
Time taken (seconds) to perform last macro:

real: 0.4200000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?
(38a) DISPLAY LATTICE END

SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> ARG1
UNIVERSAL >> EXPR
UNIVERSAL >> NAME
UNIVERSAL >> RESULT
UNIVERSAL >> GENDER
UNIVERSAL >> MEMB
UNIVERSAL >> RSLT
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> ARG2
RELATIVE >> BLOOD_RELATIVE
RELATIVE >> IN_LAW_RELATIVE
BLOOD_RELATIVE >> CHILD
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> SIBLING_IN_LAW
IN_LAW_RELATIVE >> PARENT_IN_LAW

SS Press RETURN to continue...

SS Relation Hierarchy Details

CHILD >> SIBLING_CHILD
CHILD >> BLOOD_SIBLING
CHILD >> BLOOD_PARENT
SIBLING_IN_LAW >> BROTHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW
PARENT_IN_LAW >> FATHER_IN_LAW
PARENT_IN_LAW >> MOTHER_IN_LAW

Time taken (seconds) to perform last macro:

real: 0.2300000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(39) ASSERT LATTICE (BLOOD_SIBLING) >> (BROTHER) END

SS Message From Assertion Subsystem...
SS The relation BLOOD_SIBLING has no definition
SS Do you wish to define it (Y/N)? N
The relation BROTHER is previously unknown.
Do you wish to define it (Y/N)? N

What Next?

DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL $>$ STRING
UNIVERSAL $>$ NUMERIC
UNIVERSAL $>$ INDIVIDUAL
STRING $>$ FAMILY_NAME
STRING $>$ PERSON_NAME
INDIVIDUAL $>$ PERSON
INDIVIDUAL $>$ ACT
INDIVIDUAL $>$ MALENESS
INDIVIDUAL $>$ FEMALENESS
INDIVIDUAL $>$ FAMILY
INDIVIDUAL $>$ STATE
PERSON $>$ MAN
PERSON $>$ WOMAN
STATE $>$ MARRIAGE
ACT $>$ BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL $>$ FRIEND
UNIVERSAL $>$ RELATIVE
UNIVERSAL $>$ AGNT
UNIVERSAL $>$ EXPR
UNIVERSAL $>$ GENDER
UNIVERSAL $>$ ASLIT
UNIVERSAL $>$ ARG2
UNIVERSAL $>$ LEGAL_PARTNER
UNIVERSAL $>$ RESULT
UNIVERSAL $>$ NAME
UNIVERSAL $>$ ARG1
UNIVERSAL $>$ MEMB
RELATIVE $>$ IN_LAW_RELATIVE
RELATIVE $>$ BLOOD_RELATIVE
IN_LAW_RELATIVE $>$ SPOUSE
IN_LAW_RELATIVE $>$ PARENT_IN_LAW
IN_LAW_RELATIVE $>$ SIBLING_IN_LAW
BLOOD_RELATIVE $>$ CHILD

Press RETURN to continue...

Relation Hierarchy Details

PARENT_IN_LAW $>$ MOTHER_IN_LAW
PARENT_IN_LAW $>$ FATHER_IN_LAW
SIBLING_IN_LAW $>$ SISTER_IN_LAW
SIBLING_IN_LAW $>$ BROTHER_IN_LAW
CHILD $>$ BLOOD_PARENT
CHILD $>$ BLOOD_SIBLING
CHILD $>$ SIBLING_CHILD
BLOOD_SIBLING $>$ BROTHER

Time taken (seconds) to perform last macro:

real: 0.2300000
user: 0.0090000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

ASSER LATTICE (BLOOD_SIBLING) $>$ (SISTER) END

Message From Assertion Subsystem...
The relation BLOOD_SIBLING has no definition
Do you wish to define it (Y/N)? N

The relation SISTER is previously unknown
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.3700000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(DISPLAY LATTICE END)

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL > GENDER
UNIVERSAL > RELT
UNIVERSAL > ARG2
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
UNIVERSAL > EXPR
UNIVERSAL > AGNT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

Press RETURN to continue...

Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SISTER

Time taken (seconds) to perform last macro:
real: 0.2500000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

What Next?
(41) ASSERT LATTICE (BLOOD_SIBLING) > (SIBLING_IN_LAW) END

$$ Message From Assertion Subsystem...
$$ The relation BLOOD_SIBLING has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation SIBLING_IN_LAW has no definition
$$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th>user</th>
<th>sys</th>
<th>child user</th>
<th>child sys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.440</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>

$$ What Next?

(41a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL >> RESULT
UNIVERSAL >> NAME
UNIVERSAL >> AGNT
UNIVERSAL >> MEMB
UNIVERSAL >> Expr
UNIVERSAL >> AGNT
UNIVERSAL >> RELATIVE
UNIVERSAL >> FRIEND
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> ARG2
UNIVERSAL >> REL
UNIVERSAL >> GENDER
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SIBLING_IN_LAW
BLOOD_RELATIVE >> CHILD

$$ Press RETURN to continue...

$$ Relation Hierarchy Details

PARENT_IN_LAW >> MOTHER_IN_LAW
PARENT_IN_LAW >> FATHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW
SIBLING_IN_LAW >> BROTHER_IN_LAW
CHILD >> BLOOD_SIBLING
CHILD >> SIBLING_CHILD
CHILD >> BLOOD_PARENT
BLOOD_SIBLING >> SISTER
BLOOD_SIBLING >> BROTHER
BLOOD_SIBLING >> SIBLING_IN_LAW

Time taken (seconds) to perform last macro:

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th>user</th>
<th>sys</th>
<th>child user</th>
<th>child sys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.280</td>
<td>0.010</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
What Next?

(42) ASSERT LATTICE (SIBLING_CHILD) > (NEPHEW) END

Message From Assertion Subsytem...
The relation SIBLING_CHILD has no definition
Do you wish to define it (Y/N)? N

Message From Assertion Subsytem...
The relation NEPHEW is previously unknown
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3500000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(42a) DISPLAY LATTICE END

Type Hierarchy Details

| UNIVERSAL | STRING |
| UNIVERSAL | NUMERIC |
| UNIVERSAL | INDIVIDUAL |
| STRING | FAMILY_NAME |
| STRING | COMPANY_NAME |
| STRING | PERSON_NAME |
| INDIVIDUAL | PERSON |
| INDIVIDUAL | ACT |
| INDIVIDUAL | MALENESS |
| INDIVIDUAL | FEMALENESS |
| INDIVIDUAL | FAMILY |
| INDIVIDUAL | STATE |
| PERSON | MAN |
| PERSON | WOMAN |
| STATE | MARRIAGE |
| ACT | BIRTH |

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

| UNIVERSAL | EXPR |
| UNIVERSAL | AGNT |
| UNIVERSAL | RELATIVE |
| UNIVERSAL | FRIEND |
| UNIVERSAL | LEGAL_PARTNER |
| UNIVERSAL | ARG2 |
| UNIVERSAL | RSLT |
| UNIVERSAL | GENDER |
| UNIVERSAL | MEMB |
| UNIVERSAL | ARG1 |
| UNIVERSAL | NAME |
| UNIVERSAL | RESULT |
| RELATIVE | IN_LAW_RELATIVE |
| RELATIVE | BLOOD_RELATIVE |
| IN_LAW_RELATIVE | SPOUSE |
| IN_LAW_RELATIVE | PARENT_IN_LAW |
| IN_LAW_RELATIVE | SIBLING_IN_LAW |
| BLOOD_RELATIVE | CHILD |

Press RETURN to continue...

Relation Hierarchy Details

| PARENT_IN_LAW | MOTHER_IN_LAW |
| PARENT_IN_LAW | FATHER_IN_LAW |
| SIBLING_IN_LAW | SISTER_IN_LAW |
| SIBLING_IN_LAW | BROTHER_IN_LAW |
| CHILD | BLOOD_PARENT |
| CHILD | BLOOD_SIBLING |
| CHILD | SIBLING_CHILD |
| BLOOD_SIBLING | SIBLING_IN_LAW |
| BLOOD_SIBLING | BROTHER |
| BLOOD_SIBLING | SISTER |
| SIBLING_CHILD | NEPHEW |

Time taken (seconds) to perform last macro:
SS What Next?

(43) ASSERT LATTICE  (SIBLING_CHILD) > (NEICE) END

SS Message From Assertion Subsystem...
SS The relation SIBLING_CHILD has no definition
SS Do you wish to define it (Y/N)? N

SS Message From Assertion Subsystem...
SS The relation NEICE is previously unknown
SS Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.3600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(43a) DISPLAY LATTICE END

SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
UNIVERSAL > MEMB
UNIVERSAL > ARG1
UNIVERSAL > NAME
UNIVERSAL > RESULT
UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > EXPR
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

SS Press RETURN to continue...

SS Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > NIECE
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SIBLING_IN_LAW

Time taken (seconds) to perform last macro:
real: 0.2600000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(44) ASSEMBLE LATTICE (SIBLING_CHILD) > (COUSIN_GERMAN) END

$$ Message From Assertion Subsystem...
$$ The relation SIBLING_CHILD has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation COUSIN_GERMAN is previously unknown
$$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 1.0500000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(44a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL >> MEMB
UNIVERSAL >> ARG1
UNIVERSAL >> NAME
UNIVERSAL >> RESULT
UNIVERSAL >> FRIEND
UNIVERSAL >> RELATIVE
UNIVERSAL >> AGNT
UNIVERSAL >> EXPR
UNIVERSAL >> GENDER
UNIVERSAL >> RELT
UNIVERSAL >> ARG2
UNIVERSAL >> LEGAL_PARTNER
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SIBLING_IN_LAW
BLOOD_RELATIVE >> CHILD

$$ Press RETURN to continue...

$$ Relation Hierarchy Details

PARENT_IN_LAW >> MOTHER_IN_LAW
PARENT_IN_LAW >> FATHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW
SIBLING_IN_LAW >> BROTHER_IN_LAW
$S$ Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
SIBLING_CHILD > NIECE
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > COUSIN_GERMAN
BLOOD_PARENT > PARENT_IN_LAW

Time taken (seconds) to perform last macro:

real: 0.2200000
user: 0.0300000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$S$ What Next?

(46) ASSERT LATTICE (BLOOD_PARENT) > (BLOOD_GRANDPARENT) END

$S$ Message From Assertion Subsystem...
$S$ The relation BLOOD_PARENT has no definition
$S$ Do you wish to define it (Y/N)? N

$S$ Message From Assertion Subsystem...
$S$ The relation BLOOD_GRANDPARENT is previously unknown
$S$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.4400000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$S$ What Next?

(46a) DISPLAY LATTICE END

$S$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$S$ Press RETURN to look at the relation hierarchy...

$S$ Relation Hierarchy Details

UNIVERSAL > EKPR
UNIVERSAL > AGT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
UNIVERSAL > MEMB
UNIVERSAL > ARG1
UNIVERSAL > NAME
UNIVERSAL > RESULT
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$$ Press RETURN to continue...

$$ Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
BLOOD_PARENT > FATHER
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
SIBLING_CHILD > NIECE
SIBLING_CHILD > Nephew
SIBLING_CHILD > COUSIN_GERMAN

Time taken (seconds) to perform last macro:
real: 0.2900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(47) ASSERT LATTICE (BLOOD_PARENT) > (FATHER) END

$$ Message From Assertion Subsystem...
$$ The relation BLOOD_PARENT has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation FATHER is previously unknown
$$ Do you wish to define it (Y/N)? N
Time taken (seconds) to perform last macro:
real: 0.4100000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(47a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
UNIVERSAL > EXPR
UNIVERSAL > AGNT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > RESULT
UNIVERSAL > GENDER
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$ SS Press RETURN to continue...

$ SS Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
BLOOD_SIBLING > SIBLING_IN_LAW
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
SIBLING_CHILD > NIECE
SIBLING_CHILD > COUSIN_GERMAN
SIBLING_CHILD > NEPHEW
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > FATHER

Time taken (seconds) to perform last macro:
real: 0.2700000  
user: 0.0000000  
sys: 0.0000000  
child user: 0.0000000  
child sys: 0.0000000

$ SS What Next?

(48) ASSERT LATTICE (BLOOD_PARENT) > (MOTHER) END

$ SS Message From Assertion Subsystem...
$ SS The relation BLOOD_PARENT has no definition
$ SS Do you wish to define it (Y/N)? N

$ SS Message From Assertion Subsystem...
$ SS The relation MOTHER is previously unknown
$ SS Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.3700000  
user: 0.0100000  
sys: 0.0000000  
child user: 0.0000000  
child sys: 0.0000000

$ SS What Next?

(48a) DISPLAY LATTICE END

$ SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > EXPR
UNIVERSAL > AGNT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > REL
UNIVERSAL > GENDER
UNIVERSAL > MEMB
UNIVERSAL > ARG1
UNIVERSAL > NAME
UNIVERSAL > RESULT
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$$ Press RETURN to continue...

$$ Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
BLOOD_PARENT > FATHER
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > MOTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SIBLING_CHILD
SIBLING_CHILD > NIECE
SIBLING_CHILD > NIEPHEW
SIBLING_CHILD > COUSIN

Time taken (seconds) to perform last macro:
real: 0.2900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(49) ASSERT LATTICE (BROTHER) > (BROTHER_IN_LAW) END

$$ Message From Assertion Subsystem...
$$ The relation BROTHER has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation BROTHER_IN_LAW has no definition
$$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.3900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(49a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
Universals >> Individuals
String >> Family Name
String >> Person Name
Individual >> Person
Individual >> Act
Individual >> Maleness
Individual >> Femaleness
Individual >> Family
Individual >> State
Person >> Man
Person >> Woman
State >> Marriage
Act >> Birth

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

Universal > Legal Partner
Universal > Arg2
Universal > Rslt
Universal > Geder
Universal > Mebm
Universal > Arg1
Universal > Name
Universal > Rslt
Universal > Friend
Universal > Relative
Universal > Agent
Relative > In Law Relative
Relative > Blood Relative
In Law Relative > Spouse
In Law Relative > Parent In Law
In Law Relative > Sibling In Law
Blood Relative > Child

$\$ Press RETURN to continue...

$\$ Relation Hierarchy Details

Parent In Law > Mother In Law
Parent In Law > Father In Law
Sibling In Law > Sister In Law
Sibling In Law > Brother In Law
Child > Sibling Child
Child > Blood Parent
Child > Blood Sibling
Sibling Child > Cousin German
Sibling Child > Nephew
Sibling Child > Niece
Blood Parent > Mother
Blood Parent > Parent In Law
Blood Parent > Blood Grandparent
Blood Parent > Father
Blood Sibling > Sibling In Law
Blood Sibling > Sister
Blood Sibling > Brother
Brother > Brother In Law

Time taken (seconds) to perform last macro:

real: 0.4500000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(50) ASSERT (Brother) > (Uncle) END

$\$ Message From Assertion Subsystem...

$\$ The relation Brother has no definition
$\$ Do you wish to define it (Y/N)? N

$\$ Message From Assertion Subsystem...

$\$ The relation Uncle is previously unknown
$\$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.4100000
user: 0.0000000
sys: 0.0100000
child user:  0.0000000
child sys:  0.0000000

What Next?

(50a) DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL > MEMB
UNIVERSAL > ARG1
UNIVERSAL > NAME
UNIVERSAL > RESULT
UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > EXPR
UNIVERSAL > GENDER
UNIVERSAL > RSLT
UNIVERSAL > ARG2
UNIVERSAL > LEGAL_PARTNER
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

Press RETURN to continue...

Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
SIBLING_CHILD > NIECE
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > COUSIN_GERMAN
BLOOD_PARENT > MOTHER
BLOOD_PARENT > FATHER
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > PARENT_IN_LAW
BROTHER > BROTHER_IN_LAW

Press RETURN to continue...

Relation Hierarchy Details

BROTHER > UNCLE

Time taken (seconds) to perform last macro:

real:  0.4100000
user:  0.0100000
sys:  0.0100000
child user:  0.0000000
child sys:  0.0000000
SS What Next?

(51) ASSERT LATTICE (SISTER) > (SISTER_IN_LAW) END

SS Message From Assertion Subsystem...
SS The relation SISTER has no definition
SS Do you wish to define it (Y/N)? N

SS Message From Assertion Subsystem...
SS The relation SISTER_IN_LAW has no definition
SS Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(51a) DISPLAY LATTICE END

SS Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

SS Press RETURN to look at the relation hierarchy...

SS Relation Hierarchy Details

UNIVERSAL >> GENDER
UNIVERSAL >> REL
UNIVERSAL >> ARG2
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> RESULT
UNIVERSAL >> NAME
UNIVERSAL >> ARG1
UNIVERSAL >> MEMB
UNIVERSAL >> Expr
UNIVERSAL >> AGT
UNIVERSAL >> RELATIVE
UNIVERSAL >> FRIEND
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SIBLING_IN_LAW
BLOOD_RELATIVE >> CHILD

SS Press RETURN to continue...

SS Relation Hierarchy Details

PARENT_IN_LAW >> MOTHER_IN_LAW
PARENT_IN_LAW >> FATHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW
SIBLING_IN_LAW >> BROTHER_IN_LAW
CHILD >> SIBLING_CHILD
CHILD >> BLOOD_PARENT
CHILD >> BLOOD_SIBLING
SIBLING_CHILD >> NEPHEW
SIBLING_CHILD >> COUSIN_GERMAN
SIBLING_CHILD >> NIECE
BLOOD_PARENT >> MOTHER
BLOOD_PARENT >> FATHER
BLOOD_PARENT >> BLOOD_GRANDPARENT
BLOOD_PARENT >> PARENT_IN_LAW
BLOOD_SIBLING >> BROTHER
BLOOD_SIBLING >> SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
BROTHER > BROTHER_IN_LAW

$§ Press RETURN to continue...

$§ Relation Hierarchy Details

BROTHER > UNCLE
SISTER > SISTER_IN_LAW

Time taken (seconds) to perform last macro:

real: 0.4300000
user: 0.0000000
sys: 0.0200000
child user: 0.0000000
child sys: 0.0000000

$§ What Next?

(52) ASSERT LATTICE (SISTER) > (AUNT) END

$§ Message From Assertion Subsystem...
$§ The relation SISTER has no definition
$§ Do you wish to define it (Y/N)? N

$§ Message From Assertion Subsystem...
$§ The relation AUNT is previously unknown
$§ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3900000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$§ What Next?

(52a) DISPLAY LATTICE END

$§ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$§ Press RETURN to look at the relation hierarchy...

$§ Relation Hierarchy Details

UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
UNIVERSAL > EXPR
UNIVERSAL > AGNT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$§ Press RETURN to continue...
$\$ \text{Relation Hierarchy Details}

PARENT_IN_LAW > MOTHER_IN_LAW  
PARENT_IN_LAW > FATHER_IN_LAW  
SIBLING_IN_LAW > SISTER_IN_LAW  
SIBLING_IN_LAW > BROTHER_IN_LAW  
CHILD > BLOOD_SIBLING  
CHILD > SIBLING_CHILD  
CHILD > BLOOD_PARENT  
BLOOD_SIBLING > SIBLING_IN_LAW  
BLOOD_SIBLING > SISTER  
BLOOD_SIBLING > BROTHER  
SIBLING_CHILD > NIECE  
SIBLING_CHILD > COUSIN_GERMAN  
SIBLING_CHILD > NEPHEM  
BLOOD_PARENT > MOTHER  
BLOOD_PARENT > PARENT_IN_LAW  
BLOOD_PARENT > BLOOD_GRANDPARENT  
BLOOD_PARENT > FATHER  
SISTER > SISTER_IN_LAW

$\$ \text{Press RETURN to continue...}

$\$ \text{Relation Hierarchy Details}

SISTER > AUNT  
BROTHER > UNCLE  
BROTHER > BROTHER_IN_LAW

Time taken (seconds) to perform last macro:

real: \ 0.4400000  
user: \ 0.0000000  
sys: \ 0.0000000  
child user: \ 0.0000000  
child sys: \ 0.0000000

$\$ \text{What Next?}

(53a) \text{DISPLAY LATTICE END}

$\$ \text{Type Hierarchy Details}

UNIVERSAL >> STRING  
UNIVERSAL >> NUMERIC  
UNIVERSAL >> INDIVIDUAL  
STRING >> FAMILY_NAME  
STRING >> COMPANY_NAME  
STRING >> PERSON_NAME  
INDIVIDUAL >> PERSON  
INDIVIDUAL >> ACT  
INDIVIDUAL >> MALENESS  
INDIVIDUAL >> FEMALENESS  
INDIVIDUAL >> FAMILY  
INDIVIDUAL >> STATE  
PERSON >> MAN  
PERSON >> WOMAN  
STATE >> MARRIAGE  
ACT >> BIRTH

$\$ \text{Press RETURN to look at the relation hierarchy...}

$\$ \text{Relation Hierarchy Details}
SS Press RETURN to continue...

SS Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
BLOOD_PARENT > FATHER
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > MOTHER
BLOOD_SIBLING > BROTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
SIBLING_CHILD > NIECE
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > COUSIN_GERMAN
BROTHER > BROTHER_IN_LAW

SS Press RETURN to continue...

SS Relation Hierarchy Details

BROTHER > UNCLE
SISTER > AUNT
SISTER > SISTER_IN_LAW
COUSIN_GERMAN > SECOND_COUSIN

Time taken (seconds) to perform last macro:

real: 0.4300000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(54) ASSERT LATTICE (BLOOD_GRANDPARENT) > (GRANDFATHER) END

SS Message From Assertion Subsystem...
SS The relation BLOOD_GRANDPARENT has no definition
SS Do you wish to define it (Y/N)? N

SS Message From Assertion Subsystem...
SS The relation GRANDFATHER is previously unknown
SS Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3500000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(54a) DISPLAY LATTICE END

SS Type Hierarchy Details
UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$\$ Press RETURN to look at the relation hierarchy...

$\$ Relation Hierarchy Details

UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > RSLT
UNIVERSAL > GENDER
UNIVERSAL > MEMB
UNIVERSAL > ARG1
UNIVERSAL > NAME
UNIVERSAL > RESULT
UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > EXPR
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$\$ Press RETURN to continue...

$\$ Relation Hierarchy Details

PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
SIBLING_CHILD > COUSIN_GERMAN
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > NIECE
BLOOD_PARENT > MOTHER
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > FATHER
BLOOD_SIBLING > SIBLING_IN_LAW
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
COUSIN_GERMAN > SECOND_Cousin

$\$ Press RETURN to continue...

$\$ Relation Hierarchy Details

BLOOD_GRANDPARENT > GRANDFATHER
SISTER > SISTER_IN_LAW
SISTER > AUNT
BROTHER > UNCLE
BROTHER > BROTHER_IN_LAW

Time taken (seconds) to perform last macro:

real: 0.4700000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(55) ASSERT LATTICE (BLOOD_GRANDPARENT) > (GRANDMOTHER) END

$\$ Message From Assertion Subsystem...
The relation BLOOD_GRANDPARENT has no definition
Do you wish to define it (Y/N)?  N

Message From Assertion Subsystem...
The relation GRANDMOTHER is previously unknown
Do you wish to define it (Y/N)?  N

Time taken (seconds) to perform last macro:

real: 0.3700000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

Type Hierarchy Details

DISPLAY LATTICE END

Relation Hierarchy Details

Press RETURN to look at the relation hierarchy...

Press RETURN to continue...

Relation Hierarchy Details

Press RETURN to continue...

Relation Hierarchy Details
BROTHER > UNCLE
SISTER > AUNT
SISTER > SISTER-IN-LAW
CUISIN_GERMAN > SECOND_COUSIN
BLOOD_GRANDPARENT > GRANDFATHER
BLOOD_GRANDPARENT > GRANDMOTHER

Time taken (seconds) to perform last macro:

real: 0.4700000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(56) ASSERT LATTICE (FATHER) > (FATHER_IN_LAW) END

$$ Message From Assertion Subsystem...
$$ The relation FATHER has no definition
$$ Do you wish to define it (Y/N)? N

$$ Message From Assertion Subsystem...
$$ The relation FATHER_IN_LAW has no definition
$$ Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:

real: 0.3600000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(56a) DISPLAY LATTICE END

$$ Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$$ Press RETURN to look at the relation hierarchy...

$$ Relation Hierarchy Details

UNIVERSAL > FRIEND
UNIVERSAL > RELATIVE
UNIVERSAL > AGNT
UNIVERSAL > EXPR
UNIVERSAL > GENDER
UNIVERSAL > RSLT
UNIVERSAL > ARG2
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
RELATIVE > IN_LAW_RELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAW_RELATIVE > SPOUSE
IN_LAW_RELATIVE > PARENT_IN_LAW
IN_LAW_RELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD

$$ Press RETURN to continue...

$$ Relation Hierarchy Details
PARENT_IN-LAW > MOTHER_IN-LAW
PARENT_IN-LAW > FATHER_IN-LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_PARENT
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > FATHER
BLOOD_PARENT > MOTHER
BLOOD_SIBLING > SIBLING_IN_LAW
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
SIBLING_CHILD > NIECE
SIBLING_CHILD > COUSIN_GERMAN
SIBLING_CHILD > COUSIN_GERMAN
BLOOD_GRANDPARENT > GRANDMOTHER

$S Press RETURN to continue...

$S Relation Hierarchy Details

BLOOD_GRANDPARENT > GRANDFATHER
FATHER > FATHER_IN_LAW
SISTER > SISTER_IN_LAW
SISTER > AUNT
BROTHER > UNCLE
BROTHER > BROTHER_IN_LAW
COUSIN_GERMAN > SECOND_COUSIN

Time taken (seconds) to perform last macro:
real: 0.4200000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$S What Next?

(57) ASSERT LATTICE (FATHER) > (GRANDFATHER) END

$S Message From Assertion Subsystem...
$S The relation FATHER has no definition
$S Do you wish to define it (Y/N)? N

$S Message From Assertion Subsystem...
$S The relation GRANDFATHER has no definition
$S Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.4200000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$S What Next?

(57a) DISPLAY LATTICE END

$S Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

$S Press RETURN to look at the relation hierarchy...
$$ What Next?$$

(56a) DISPLAY LATTICE END

$$ Type Hierarchy Details $$

```
UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH
```

$$ Press RETURN to look at the relation hierarchy...$$

$$ Relation Hierarchy Details $$

```
UNIVERSAL > RESULT
UNIVERSAL > NAME
UNIVERSAL > ARG1
UNIVERSAL > MEMB
UNIVERSAL > EXPR
UNIVERSAL > AGNT
UNIVERSAL > RELATIVE
UNIVERSAL > FRIEND
UNIVERSAL > LEGAL_PARTNER
UNIVERSAL > ARG2
UNIVERSAL > REL
UNIVERSAL > GENDER
RELATIVE > IN_LAWRELATIVE
RELATIVE > BLOOD_RELATIVE
IN_LAWRELATIVE > SPOUSE
IN_LAWRELATIVE > PARENT_IN_LAW
IN_LAWRELATIVE > SIBLING_IN_LAW
BLOOD_RELATIVE > CHILD
```

$$ Press RETURN to continue...$$

$$ Relation Hierarchy Details $$

```
PARENT_IN_LAW > MOTHER_IN_LAW
PARENT_IN_LAW > FATHER_IN_LAW
SIBLING_IN_LAW > SISTER_IN_LAW
SIBLING_IN_LAW > BROTHER_IN_LAW
CHILD > BLOOD_SIBLING
CHILD > SIBLING_CHILD
CHILD > BLOOD_PARENT
BLOOD_SIBLING > SIBLING_IN_LAW
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > BROTHER
SIBLING_CHILD > NIECE
SIBLING_CHILD > COUSIN_GERMAN
SIBLING_CHILD > NEPHEW
BLOOD_PARENT > MOTHER
BLOOD_PARENT > PARENT_IN_LAW
BLOOD_PARENT > BLOOD_GRANDPARENT
BLOOD_PARENT > FATHER
SISTER > SISTER_IN_LAW
```

$$ Press RETURN to continue...$$

$$ Relation Hierarchy Details $$

```
SISTER > AUNT
BROTHER > UNCLE
BROTHER > BROTHER_IN_LAW
COUSIN_GERMAN > SECOND_COUSIN
MOTHER > MOTHER_IN_LAW
BLOOD_GRANDPARENT > GRANDMOTHER
BLOOD_GRANDPARENT > GRANDFATHER
FATHER > FATHER_IN_LAW
FATHER > GRANDFATHER
```

Time taken (seconds) to perform last macro:
real: 0.4500000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

What Next?

(59) ASSERT LATTICE (MOTHER) > (GRANDMOTHER) END

Message From Assertion Subsystem...
The relation MOTHER has no definition
Do you wish to define it (Y/N)? N

Message From Assertion Subsystem...
The relation GRANDMOTHER has no definition
Do you wish to define it (Y/N)? N

Time taken (seconds) to perform last macro:
real: 0.3800000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(59a) DISPLAY LATTICE END

Type Hierarchy Details

UNIVERSAL >> STRING
UNIVERSAL >> NUMERIC
UNIVERSAL >> INDIVIDUAL
STRING >> FAMILY_NAME
STRING >> COMPANY_NAME
STRING >> PERSON_NAME
INDIVIDUAL >> PERSON
INDIVIDUAL >> ACT
INDIVIDUAL >> MALENESS
INDIVIDUAL >> FEMALENESS
INDIVIDUAL >> FAMILY
INDIVIDUAL >> STATE
PERSON >> MAN
PERSON >> WOMAN
STATE >> MARRIAGE
ACT >> BIRTH

Press RETURN to look at the relation hierarchy...

Relation Hierarchy Details

UNIVERSAL >> EXPR
UNIVERSAL >> AGNT
UNIVERSAL >> RELATIVE
UNIVERSAL >> FRIEND
UNIVERSAL >> LEGAL_PARTNER
UNIVERSAL >> ARG2
UNIVERSAL >> RSLT
UNIVERSAL >> GENDER
UNIVERSAL >> HEMS
UNIVERSAL >> ARG1
UNIVERSAL >> NAME
UNIVERSAL >> RESULT
RELATIVE >> IN_LAW_RELATIVE
RELATIVE >> BLOOD_RELATIVE
IN_LAW_RELATIVE >> SPOUSE
IN_LAW_RELATIVE >> PARENT_IN_LAW
IN_LAW_RELATIVE >> SIBLING_IN_LAW
BLOOD_RELATIVE >> CHILD

Press RETURN to continue...

Relation Hierarchy Details

PARENT_IN_LAW >> MOTHER_IN_LAW
PARENT_IN_LAW >> FATHER_IN_LAW
SIBLING_IN_LAW >> SISTER_IN_LAW
SIBLING_IN_LAW >> BROTHER_IN_LAW
CHILD >> BLOOD_PARENT
CHILD >> BLOOD_SIBLING
CHILD >> SIBLING_CHILD
BLOOD_PARENT >> FATHER
BLOOD_PARENT >> BLOOD_GRANDPARENT
BLOOD_PARENT >> PARENT_IN_LAW
BLOOD_PARENT >> MOTHER
BLOOD_SIBLING >> BROTHER
BLOOD_SIBLING > SISTER
BLOOD_SIBLING > SIBLING_IN_LAW
SIBLING_CHILD > NIECE
SIBLING_CHILD > NEPHEW
SIBLING_CHILD > COUSIN_GERMAN
FATHER > GRANDFATHER

$\$ Press RETURN to continue...

$\$ Relation Hierarchy Details

FATHER > FATHER_IN_LAW
BLOOD_GRANDPARENT > GRANDFATHER
BLOOD_GRANDPARENT > GRANDMOTHER
MOTHER > MOTHER_IN_LAW
MOTHER > GRANDMOTHER
BROTHER > BROTHER_IN_LAW
BROTHER > UNCLE
SISTER > AUNT
SISTER > SISTER_IN_LAW
COUSIN_GERMAN > SECOND_COUSIN

Time taken (seconds) to perform last macro:
real: 0.4500000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(60) ASSERT CATALOGUE
    [FAMILY] -
    (NAME)->[FAMILY_NAME]
    (MEMB)->[PERSON] -
    (RESULT)<-[ACT].

END

Time taken (seconds) to perform last macro:
real: 1.3800000
user: 0.0600000
sys: 0.0300000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(60a) DISPLAY CATALOGUE END

[FAMILY] -
    (NAME)->[FAMILY_NAME]
    (MEMB)->[PERSON] -
    (RESULT)<-[EVENT],

Time taken (seconds) to perform last macro:
real: 0.0200000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(61) ASSERT CATALOGUE
    [PERSON] -
    (RELATIVE)->[PERSON]
    (FRIEND)->[PERSON]
    (NAME)->[PERSON_NAME]
    (GENDER)->[MALENESS].

END

Time taken (seconds) to perform last macro:
real: 2.1000000
user: 0.1000000
sys: 0.0200000
child user: 0.0000000
child sys: 0.0000000
(61a) DISPLAY CATALOGUE END

[FAMILY] -
(NAME) -> [FAMILY_NAME]
(MEMBER) -> [PERSON] -
(RESULT) <- [EVENT]
_RELATIVITY) -> [PERSON]
(FRIEND) -> [PERSON]
(NAME) -> [PERSON_NAME]
(GENDER) -> [MALENESS],

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(62) ASSERT CATALOGUE

[PERSON] -
(GENDER) -> [FEMALENESS]
(EXPR) <- [STATE]
(AGNT) <- [ACT]
(LEGAL_PARTNER) -> [PERSON].
END

Time taken (seconds) to perform last macro:
real: 5.8600000
user: 0.2500000
sys: 0.0700000
child user: 0.0000000
child sys: 0.0000000

(62a) DISPLAY CATALOGUE END

[FAMILY] -
(NAME) -> [FAMILY_NAME]
(MEMBER) -> [PERSON] -
(RESULT) <- [EVENT]
_RELATIVITY) -> [PERSON]
(FRIEND) -> [PERSON]
(NAME) -> [PERSON_NAME]
(GENDER) -> [MALENESS]
(EXPR) <- [STATE]
(AGNT) <- [ACT]
(LEGAL_PARTNER) -> [PERSON].

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(63) ASSERT TYPE MAN (x)

[PERSON: *x] -> [GENDER] -> [MALENESS].
END

Time taken (seconds) to perform last macro:
real: 0.2200000
user: 0.0200000
sys: 0.0200000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?
(63a) DISPLAY TYPE MAN END
TYPE MAN (*104) is
{PERSON : *104} -
{GENDER} -> {MALENESS : *105}

Time taken (seconds) to perform last macro:
real: 0.0400000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?
(64) ASSERT TYPE WOMAN (x)
{PERSON : *x} -> {GENDER} -> {FEMALENESS}.
END

Time taken (seconds) to perform last macro:
real: 0.0400000
user: 0.0300000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?
(64a) DISPLAY TYPE WOMAN END
TYPE WOMAN (*123) is
{PERSON : *123} -
{GENDER} -> {FEMALENESS : *124}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?
(65) ASSERT TYPE BIRTH (x)
{ACT : *x} -> {RESULT} -> {PERSON}.
END

Time taken (seconds) to perform last macro:
real: 0.0400000
user: 0.0300000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?
(65a) DISPLAY TYPE BIRTH END
TYPE BIRTH (*142) is
{EVENT : *142} -
{RESULT} -> {PERSON : *143}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?
(66) ASSERT TYPE MARRIAGE (x)
     [STATE:*x] -
     (EXPR)->[PERSON:*a] -
     (LEGAL_PARTNER)->[PERSON:*b],
     (EXPR)->[PERSON:*b] -
     (LEGAL_PARTNER)->[PERSON:*a],

Time taken (seconds) to perform last macro:
real: 0.0800000
user: 0.0800000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(66a) DISPLAY TYPE MARRIAGE END

TYPE MARRIAGE (*161) is

     [STATE : *161] -
     (EXPR)->[PERSON : *162] -
     (LEGAL_PARTNER)->[PERSON : *163] -
     (EXPR)->[STATE : *161] -
     (LEGAL_PARTNER)->[PERSON : *162],

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(67) ASSERT RELATION BLOOD_RELATIVE (x,y)
     [FAMILY] -
     (MEMB)-> [PERSON: *x]
     (MEMB)-> [PERSON: *y].

Time taken (seconds) to perform last macro:
real: 0.0700000
user: 0.0500000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(67a) DISPLAY RELATION BLOOD_RELATIVE END

RELATION BLOOD_RELATIVE (*184, *185) is

     [FAMILY : *183] -
     (MEMB)->[PERSON : *184] -
     (MEMB)->[PERSON : *185]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(68) ASSERT CATALOGUE
     [PERSON]>({BLOOD_RELATIVE})->[PERSON]

END

Time taken (seconds) to perform last macro:
real: 2.3900000
user: 0.2200000
sys: 0.0400000
child user:  0.0000000
child sys:  0.0000000

SS What Next?

(68a) DISPLAY CATALOGUE END

[STATE] -
  [EXPR] - [PERSON] -
  [GENDER] - [FEMALENESS]
  [RELATIVE] - [PERSON]
  [LEGAL_PARTNER] - [PERSON]
  [MEMB] - [FAMILY] -
  [NAME] - [FAMILY_NAME],
  [RESULT] - [ACT]
  [FRIEND] - [PERSON]
  [AGNT] - [ACT]
  [GENDER] - [MALENESS]
  [NAME] - [PERSON_NAME]
  [BLOOD_RELATIVE] - [PERSON].

Time taken (seconds) to perform last macro:

real:  0.0000000
user:  0.0000000
sys:   0.0000000
child user:  0.0000000
child sys:  0.0000000

SS What Next?

(69) ASSERT RELATION CHILD (x,y)

[PERSON: *y] -
  [BLOOD_RELATIVE] - [PERSON] -
  [AGNT] - [BIRTH: *a],
  [RESULT] - [BIRTH: *a],
  [BLOOD_RELATIVE] - [PERSON: *x] -
  [AGNT] - [BIRTH: *a].

END

Time taken (seconds) to perform last macro:

real:  0.1100000
user:  0.1000000
sys:   0.0000000
child user:  0.0000000
child sys:  0.0000000

SS What Next?

(69a) DISPLAY RELATION CHILD END

RELATION CHILD (*267, *264) is

[PERSON: *264] -
  [BLOOD_RELATIVE] - [PERSON: *265] -
  [AGNT] - [BIRTH: *266] -
  [AGNT] - [PERSON: *267] -
  [BLOOD_RELATIVE] - [PERSON: *264].
  [RESULT] - [PERSON: *264].

Time taken (seconds) to perform last macro:

real:  0.0000000
user:  0.0000000
sys:   0.0000000
child user:  0.0000000
child sys:  0.0000000

SS What Next?

(70) ASSERT RELATION SPOUSE (x,y)

[MARRIAGE] -
  [EXPR] - [PERSON: *x]
  [EXPR] - [PERSON: *y].

END

Time taken (seconds) to perform last macro:

real:  0.0700000
user:  0.0600000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

SS What Next?

(70a) DISPLAY RELATION SPOUSE END

RELATION SPOUSE (*286, *289) is

[MARRIAGE : *288] -
  (EXPR)->[PERSON : *288]
  (EXPR)->[PERSON : *289]

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

SS What Next?

(71) ASSERT CATALOGUE
    [PERSON] -> (RELATIVE) -> (PERSON)
    (FRIEND) -> (PERSON)
    (GENDER) -> (FEMALENESS)
    (RESULT) <-> (ACT)
    (NAME) -> (PERSON_NAME)
    (BLOOD_RELATIVE) -> (PERSON)
    (GENDER) -> (MALENESS)
    (LEGAL_PARTNER) -> (PERSON)
    (MEMB) <-> (FAMILY)
    (NAME) -> (FAMILY_NAME),
    (EXPR) <-> (STATE)
    (AGNT) <-> (ACT)
    (SPOUSE) -> (PERSON)

Time taken (seconds) to perform last macro:

real: 1.450000
user: 0.360000
sys: 0.060000
child user: 0.000000
child sys: 0.000000

SS What Next?

(71a) DISPLAY CATALOGUE END

[PERSON] -
  (RELATIVE) -> (PERSON)
  (FRIEND) -> (PERSON)
  (GENDER) -> (FEMALENESS)
  (RESULT) <-> (ACT)
  (NAME) -> (PERSON_NAME)
  (BLOOD_RELATIVE) -> (PERSON)
  (GENDER) -> (MALENESS)
  (LEGAL_PARTNER) -> (PERSON)
  (MEMB) <-> (FAMILY)
  (NAME) -> (FAMILY_NAME),
  (EXPR) <-> (STATE)
  (AGNT) <-> (ACT)
  (SPOUSE) -> (PERSON)

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

SS What Next?

(72) ASSERT RELATION IN_LAW_RELATIVE (x, y)
    [PERSON] -
      (BLOOD_RELATIVE) -> [PERSON: *y]
      (SPOUSE) <-> [PERSON: *x].

END

Time taken (seconds) to perform last macro:

real: 0.060000
user: 0.040000
sys: 0.010000
child user: 0.000000
child sys: 0.000000

SS What Next?
(72a) DISPLAY RELATION IN_LAW_RELATIVE END

RELATION IN_LAW_RELATIVE (*316, *315) is

[PERSON : *314] -
  (BLOOD_RELATIVE)->[PERSON : *315]
  (SPOUSE)<-[PERSON : *316]

Time taken (seconds) to perform last macro:
  real: 0.0000000
  user: 0.0000000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

$$ What Next? 

(73) ASSERT CATALOGUE [PERSON]->(CHILD)->[PERSON] END

Time taken (seconds) to perform last macro:
  real: 5.7800000
  user: 0.8700000
  sys: 0.0800000
  child user: 0.0000000
  child sys: 0.0000000

$$ What Next?

(73a) DISPLAY CATALOGUE END

[ACT] -
  (AGENT)->[PERSON] -
  (LEGAL_PARTNER)->[PERSON]
  (GENDER)->[MALENESS]
  (GENDER)->[FEMALENESS]
  (FRIEND)->[PERSON]
  (RELATIVE)->[PERSON]
  (EXPR)<-[STATE]
  (MEMBER)->[FAMILY] -
    (NAME)->[FAMILY_NAME],
  (SPOUSE)->[PERSON]
  (RESULT)<-[ACT]
  (NAME)->[PERSON_NAME]
  (BLOOD_RELATIVE)->[PERSON]
  (CHILD)->[PERSON].

Time taken (seconds) to perform last macro:
  real: 0.0000000
  user: 0.0000000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

$$ What Next?

(74) ASSERT RELATION BLOOD_SIBLING (x,y)

[PERSON] -
  (CHILD)->[PERSON: *x]
  (CHILD)->[PERSON: *y].

END

Time taken (seconds) to perform last macro:
  real: 0.0500000
  user: 0.0400000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

$$ What Next?

(74a) DISPLAY RELATION BLOOD_SIBLING END

RELATION BLOOD_SIBLING (*348, *349) is
[PERSON : *347] -
(CHILD) -> [PERSON : *348]
(CHILD) -> [PERSON : *349]

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(75) ASSERT CATALOGUE

[PERSON] - (BLOOD_SIBLING) -> [PERSON]

END

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(75a) DISPLAY CATALOGUE END

[PERSON] -

(SPOUSE) -> [PERSON]

(GENDER) -> [FEMALINESS]

(GENDER) -> [MALENESS]

(AGENT) <- [ACT]

(MEMBER) <- [FAMILY] -

(NAME) -> [FAMILY_NAME]

(EXPR) <- [STATE]

(RELATIVE) -> [PERSON]

(FRIEND) -> [PERSON]

(NAME) -> [PERSON_NAME]

(CHILD) -> [PERSON]

(LEGAL_PARTNER) -> [PERSON]

(BLOOD_RELATIVE) -> [PERSON]

(RESULT) <- [ACT]

(BLOOD_SIBLING) -> [PERSON]

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(76) ASSERT RELATION BROTHER (x,y)

[PERSON: *x] - (BLOOD_SIBLING) -> [MAN: *y].

END

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(76a) DISPLAY RELATION BROTHER END

RELATION BROTHER (*393, *394) is

[PERSON: *393] -

(BLOOD_SIBLING) -> [MAN: *394]

Time taken (seconds) to perform last macro:
$\$ What Next?

(77) ASSERT RELATION SISTER (x,y) [PERSON: °x] - (BLOOD_SIBLING)-[WOMAN: °y].

END

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(77a) DISPLAY RELATION SISTER END

RELATION SISTER (*412, *413) is

[PERSON: °412] - (BLOOD_SIBLING)--[WOMAN: °413]

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(78) ASSERT RELATION SIBLING_CHILD (x,y) [PERSON] - (BLOOD_SIBLING)<-[PERSON: °x] (CHILD)->[PERSON: °y].

END

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(78a) DISPLAY RELATION SIBLING_CHILD END

RELATION SIBLING_CHILD (*432, *433) is


Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(79) ASSERT RELATION SIBLING_IN_LAW (x,y) [PERSON] - (SPOUSE)<-[PERSON: °x] (BLOOD_SIBLING)->[PERSON: °y].

END
Time taken (seconds) to perform last macro:

real: 0.0700000
user: 0.0500000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(79a) DISPLAY RELATION SIBLING_IN_LAW END

RELATION SIBLING_IN_LAW (*452, *453) is

[PERSON : *451] -
(SPOUSE) -> [PERSON : *452]
(BLOOD_SIBLING) -> [PERSON : *453]

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(80) ASSERT CATALOGUE

[PERSON] -> [SIBLING_IN_LAW] -> [PERSON].

END

Time taken (seconds) to perform last macro:

real: 5.2600000
user: 1.6300000
sys: 0.0600000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(80a) DISPLAY CATALOGUE END

(Act) -

(Result) -> [PERSON] -
(REALTIVE) -> [PERSON]
(FRIEND) -> [PERSON]
(AGENT) <- [ACT]
(MEMBER) <- [FAMILY] -
(NAME) -> [FAMILY_NAME],
(LEGAL_PARTNER) -> [PERSON]
(EXPR) <- [STATE]
(SPOUSE) -> [PERSON]
(BLOOD_SIBLING) -> [PERSON]
(NAME) -> [PERSON_NAME]
(GENDER) -> [FEMALENESS]
(GENDER) -> [MALENESS]
(CHILD) -> [PERSON]
(BLOOD_RELATIVE) -> [PERSON]
(SIBLING_IN_LAW) -> [PERSON],

Time taken (seconds) to perform last macro:

real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(81) ASSERT RELATION BROTHER_IN_LAW (x, y)

[PERSON : *x] - (SIBLING_IN_LAW) -> [PERSON : *y].

END

Time taken (seconds) to perform last macro:

real: 0.0500000
(81a) DISPLAY RELATION BROTHER_IN_LAW END

RELATION BROTHER_IN_LAW (*532, *533) is

[PERSON : *532] -
 (SIBLING_IN_LAW)->[MAN : *533]

Time taken (seconds) to perform last macro:

- real: 0.0000000
- user: 0.0000000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

(82) ASSERT RELATION SISTER_IN_LAW (x, y)

[PERSON: *x] -
 (SIBLING_IN_LAW)->[WOMAN: *y].

END

Time taken (seconds) to perform last macro:

- real: 0.0500000
- user: 0.0300000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

(82a) DISPLAY RELATION SISTER_IN_LAW END

RELATION SISTER_IN_LAW (*551, *552) is

[PERSON : *551] -
 (SIBLING_IN_LAW)->[WOMAN : *552]

Time taken (seconds) to perform last macro:

- real: 0.0000000
- user: 0.0000000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

(83) ASSERT CATALOGUE

[PERSON] ->[BROTHER] ->[MAN]

END

Time taken (seconds) to perform last macro:

- real: 11.7800000
- user: 5.9300000
- sys: 0.1500000
- child user: 0.0000000
- child sys: 0.0000000

(83a) DISPLAY CATALOGUE END

[PERSON] -
 (SIBLING_IN_LAW)->[PERSON]
 (BLOOD_RELATIVE)->[PERSON]
 (EXPR)<-[STATE]
 (BLOOD_SIBLING)->[PERSON]
 (RESULT)<-[ACT]
 (GENRE)-->[FEMALENESS]
 (MEMB)<-[FAMILY] -
(NAME) -> [FAMILY_NAME],
(GENDER) -> [MALENESS]
(AGENT) -> [ACT]
(CHILD) -> [PERSON]
(RELATIVE) -> [PERSON]
(LEGAL_PARTNER) -> [PERSON]
(FRIEND) -> [PERSON]
(NAME) -> [PERSON_NAME]
(SPouse) -> [PERSON]
(BROTher) -> [MAN]
(Sister) -> [WOMAN]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(84) ASSERT RELATION UNCLE (x, y)
    [PERSON] -
      (CHILD) -> [PERSON : *x]
      (BROTHER) -> [MAN : *y].
    END

Time taken (seconds) to perform last macro:
real: 0.0700000
user: 0.0600000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(84a) DISPLAY RELATION UNCLE END

RELATION UNCLE (*646, *647) is

[PERSON : *645] -
  (CHILD) -> [PERSON : *646]
  (BROTHER) -> [MAN : *647]

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(85) ASSERT RELATION AUNT (x, y)
    [PERSON] -
      (CHILD) -> [PERSON : *x]
      (SISTER) -> [WOMAN : *y].
    END

Time taken (seconds) to perform last macro:
real: 0.0700000
user: 0.0500000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(85a) DISPLAY RELATION AUNT END

RELATION AUNT (*666, *667) is

[PERSON : *665] -
  (CHILD) -> [PERSON : *666]
  (SISTER) -> [WOMAN : *667]
Time taken (seconds) to perform last macro:

real:  0.0100000
user:  0.0000000
sys:   0.0100000
child user:  0.0000000
child sys:   0.0000000

```
(86) ASSERT CATALOGUE
   [PERSON]->(SIBLING_CHILD)->[PERSON]
END
```

Time taken (seconds) to perform last macro:

real:  8.1000000
user:  5.1100000
sys:   0.0400000
child user:  0.0000000
child sys:   0.0000000

```
[PERSON] -
(BLOOD_SIBLING)->[PERSON]
(CITALD->[PERSON]
(LEGAL_PARTNER)->[PERSON]
(GENDER)->[FEMALENESS]
(BROTHER)->[MAN]
(RELATION)->[ACT]
(SPouse)->[PERSON]
(SIBLING_IN_LAW)->[PERSON]
(FRIEND)->[PERSON]
(RELATIVE)->[PERSON]

(AGNT)->[ACT]
(NAME)->[FAMILY_NAME],
(STATE)->[PERSON_NAME]
(GENDER)->[MALENESS]
(SISTIST)->[WOMAN]
(BLOOD_RELATIVE)->[PERSON]
(SIBLING_CHILD)->[PERSON]
```

```
(87) DISPLAY CATALOGUE END
```

```
RELATION NEPHEW (*779, *780)
```

```
[PERSON: *779] -
(CITALD CHILD)->[MAN: *780].
```

Time taken (seconds) to perform last macro:

real:  0.0100000
user:  0.0100000
sys:   0.0000000
child user:  0.0000000
child sys:   0.0000000

```
(87a) DISPLAY RELATION NEPHEW END
```

```
RELATION NEPHEW (*779, *780) is
```

```
[PERSON : *779] -
(CITALD CHILD)->[MAN : *780]
```

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$ What Next?
(88) ASSERT RELATION NIECE (*x, y)

[PERSON: *x] - (SIBLING_CHILD) -> [PERSON: *y].
END

Time taken (seconds) to perform last macro:
real: 0.050000
user: 0.050000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?
(88a) DISPLAY RELATION NIECE END

RELATION NIECE (*781, *782) is

[PERSON: *781] - (SIBLING_CHILD) -> [PERSON: *782]

Time taken (seconds) to perform last macro:
real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?
(89) ASSERT RELATION COUSIN_GERMAN (*x, y)

[PERSON: *x] - (SIBLING_CHILD) -> [PERSON: *x] (CHILD) -> [PERSON: *y].
END

Time taken (seconds) to perform last macro:
real: 0.070000
user: 0.040000
sys: 0.020000
child user: 0.000000
child sys: 0.000000

$ What Next?
(90) DISPLAY RELATION COUSIN_GERMAN END

RELATION COUSIN_GERMAN (*801, *802) is

[PERSON: *801] - (SIBLING_CHILD) -> [PERSON: *801] (CHILD) -> [PERSON: *802]

Time taken (seconds) to perform last macro:
real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$ What Next?
Time taken (seconds) to perform last macro:

real: 18.1400000
user: 12.4600000
sys: 0.0600000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(90a) DISPLAY CATALOGUE END

[PERSON] -
(SIBLING_CHILD) -> [PERSON]
(FRIEND) -> [PERSON]
(AGENT) < [ACT]
(BLOOD_RELATIVE) -> [PERSON]
(RELATIVE) -> [PERSON]
(GENDER) -> [FEMALINESS]
(CHILD) -> [PERSON]
(BLOOD_SIBLING) -> [PERSON]
(GENDER) -> [MALENESS]
(RESP) < [ACT]
(SPOUSE) -> [PERSON]
(SIBLING_IN_LAW) -> [PERSON]
(MEMB) < [FAMILY] -
(NAME) -> [FAMILY_NAME],
(EXPR) < [STATE]
(CHR) -> [PERSON]
(NAME) -> [PERSON_NAME]
(BLOOD_SIBLING) -> [PERSON]
(GENDER) -> [MALENESS]
(RESP) < [ACT]
(SPOUSE) -> [PERSON]
(SIBLING_IN_LAW) -> [PERSON]
(MEMB) < [FAMILY] -
(NAME) -> [FAMILY_NAME],
(EXPR) < [STATE]
(CHR) -> [PERSON]
(NAME) -> [PERSON_NAME]
(BLOOD_SIBLING) -> [PERSON]
(GENDER) -> [FEMALINESS]

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(91) ASSERT RELATION SECOND_COUSIN (x,y) [PERSON] -
(COUSIN_GERMAN) < [PERSON: *x]
(CHILD) -> [PERSON: *y].

END

Time taken (seconds) to perform last macro:

real: 0.0700000
user: 0.0600000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(91a) DISPLAY RELATION END

RELATION SECOND_COUSIN (*942, *943) is

[PERSON : *941] -
(COUSIN_GERMAN) < [PERSON : *942]
(CHILD) -> [PERSON : *943]

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(92) ASSERT RELATION BLOOD_PARENT (x,y) [PERSON: *y] -
(CHILD) -> [PERSON: *x].
END

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$\$ What Next?

(92a) DISPLAY RELATION BLOOD_PARENT END

RELATION BLOOD_PARENT (*945, *944) is

[PERSON : *944] -
   (CHILD)->[PERSON : *945]

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$\$ What Next?

(93) ASSERT CATALOGUE

[PERSON] -> (BLOOD_PARENT) -> [PERSON]
END

Time taken (seconds) to perform last macro:

real: 19.660000
user: 14.100000
sys: 0.070000
child user: 0.000000
child sys: 0.000000

$\$ What Next?

(93a) DISPLAY CATALOGUE END

[PERSON] -
   (BROTHER)->[MAN]
   (SISTER)->[WOMAN]
   (EXPR)<-[STATE]
   (RESULT)<-[ACT]
   (GENDER)<-[MALENESS]
   (BLOOD_SIBLING)->[PERSON]
   (CHILD)->[PERSON]
   (GENDER)->[FEMALENESS]
   (RELATIVE)<-[PERSON]
   (LEGAL_PARTNER)->[PERSON]
   (AGNT)<-[ACT]
   (FRIEND)->[PERSON]
   (SIBLING_CHILD)->[PERSON]
   (NAME)<-[PERSON_NAME]
   (COUSIN_GERMAN)->[PERSON]
   (SPouse)->[PERSON]
   (SIBLING_IN_LAW)->[PERSON]
   (MEMBER)<-[FAMILY]
   (NAME)<-[FAMILY_NAME],
   (BLOOD_PARENT)->[PERSON]

Time taken (seconds) to perform last macro:

real: 0.000000
user: 0.000000
sys: 0.000000
child user: 0.000000
child sys: 0.000000

$\$ What Next?

(94) ASSERT RELATION PARENT_IN_LAW (x, y)

[PERSON] -
   (SPouse)<-[PERSON: *x]
END

Time taken (seconds) to perform last macro:
real: 0.0500000
user: 0.0500000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(95a) DISPLAY RELATION FATHER END

RELATION FATHER (*1093, *1094) is

[PERSON : *1093] -
(BLOOD_PARENT)->[PERSON : *1094]

Time taken (seconds) to perform last macro:
real: 0.0500000
user: 0.0500000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ \text{What Next?} $$

(95b) DISPLAY RELATION MOTHER END

RELATION MOTHER (*1095, *1096) is

[PERSON : *1095] -
(BLOOD_PARENT)->[WOMAN : *1096].
Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{What Next?}$

(97) \text{ASSERT RELATION BLOOD_GRANDPARENT (x,y)}

\begin{verbatim}
[PERSON] - (BLOOD_PARENT)->[PERSON: *y] (CHILD)->[PERSON: *x].
\end{verbatim}

Time taken (seconds) to perform last macro:
real: 0.0700000
user: 0.0600000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{What Next?}$

(97a) \text{DISPLAY RELATION BLOOD_GRANDPARENT END}

RELATION BLOOD_GRANDPARENT (*1116. *1115) is

\begin{verbatim}
[PERSON] - (BLOOD_PARENT)->[PERSON: *1115] (CHILD)->[PERSON: *1116]
\end{verbatim}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\text{What Next?}$

(98) \text{ASSERT CATALOGUE}

\begin{verbatim}
[PERSON] -> (PARENT_IN_LAW)-> [PERSON] END
\end{verbatim}

Time taken (seconds) to perform last macro:
real: 20.7900000
user: 14.4200000
sys: 0.0300000
child user: 0.0000000
child sys: 0.0000000

$\text{What Next?}$

(98a) \text{DISPLAY CATALOGUE END}

\begin{verbatim}
[PERSON] -> [LEGAL_PARTNER]->[PERSON] [CHILD]->[PERSON] [NAME]->[PERSON:NAME] (RESULT)<-[ACT] [SIBLING_CHILD]->[PERSON] [GENDER]->[MALENESS] [SISTER]->[WOMAN] [AGENT]<-[ACT] [EXPR]<-[STATE] [SPUSUE]->[PERSON] [BLOOD_RELATIVE]->[PERSON] [COUSIN_GERMAN]->[PERSON] [RELATIVE]->[PERSON] [BLOOD_PARENT]->[PERSON] [MEMB]<-[FAMILY] -
\end{verbatim}
(NAME) -> [FAMILY_NAME],
(BROTHER) -> [MAN]
(FRIEND) -> [PERSON]
(SIBLING_IN_LAW) -> [PERSON]
(GENDER) -> [FEMALENESS]
(BLOOD_SIBLING) -> [PERSON]
(PARENT_IN_LAW) -> [PERSON]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

%% What Next?

(99) ASSERT RELATION FATHER_IN_LAW (x,y)
    [PERSON: *x] -
    (PARENT_IN_LAW) -> [MAN: *y].

END

Time taken (seconds) to perform last macro:
real: 0.0600000
user: 0.0500000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

%% What Next?

(99) DISPLAY RELATION FATHER_IN_LAW END

RELATION FATHER_IN_LAW (*1255, *1256) is
[PERSON: *1255] -
(PARENT_IN_LAW) -> [MAN: *1256]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

%% What Next?

(100) ASSERT RELATION MOTHER_IN_LAW (x,y)
    [PERSON: *x] -
    (PARENT_IN_LAW) -> [WOMAN: *y].

END

Time taken (seconds) to perform last macro:
real: 0.0500000
user: 0.0400000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

%% What Next?

(100a) DISPLAY RELATION MOTHER_IN_LAW END

RELATION MOTHER_IN_LAW (*1257, *1258) is
[PERSON: *1257] -
(PARENT_IN_LAW) -> [WOMAN: *1258]

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000
What Next?

(101) ASSERT CATALOGUE

END

Time taken (seconds) to perform last macro:
real: 35.8600000
user: 15.9000000
sys: 0.1300000
child user: 0.0000000
child sys: 0.0000000

What Next?

(101a) DISPLAY CATALOGUE END

[PERSON] -
[LEGAL_PARTNER] -> [PERSON]
[CHILD] -> [PERSON]
(NAME) -> [PERSON_NAME]
RESULT) -> [ACT]
[SIBLING_CHILD] -> [PERSON]
(GENDER) -> [SEX]
[SIBLING] -> [PERSON]
[AGNT] -> [ACT]
[EVR] -> [STATE]
[SPouse] -> [PERSON]
[BLOOD_RELATIVE] -> [PERSON]
[Cousin] -> [PERSON]
(RELATIVE) -> [PERSON]
(BLOOD_PARENT) -> [PERSON]
(MEMB) -> [FAMILY]
(NAME) -> [PERSON_NAME]
(BROTHER) -> [MAN]
(FRIEND) -> [PERSON]
[SIBLING_IN_LAW] -> [PERSON]
(GENDER) -> [SEX]
[BLOOD_SIBLING] -> [PERSON]
[PARENT_IN_LAW] -> [PERSON]
(BLOOD_GRANDPARENT) -> [PERSON]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(102) ASSERT RELATION GRANDFATHER (x,y)

END

Time taken (seconds) to perform last macro:
real: 0.0600000
user: 0.0500000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

What Next?

(102a) DISPLAY RELATION GRANDFATHER END

RELATION GRANDFATHER (*1427, *1428) is
[PERSON : *1427] -
(BLOOD_GRANDPARENT) -> [MAN : *1428]

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
(103) ASSERT RELATION GRANDMOTHER (x,y)
    [PERSON: *x] -
    (BLOOD_GRANDPARENT) -> [WOMAN: *y].
END

Time taken (seconds) to perform last macro:
real: 0.0700000
user: 0.0600000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(103a) DISPLAY RELATION GRANDMOTHER END

RELATION GRANDMOTHER (*1446, *1447) is
[PERSON : *1446] -
    (BLOOD_GRANDPARENT) -> [WOMAN : *1447]

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

(104) ASSERT GRAPH
    {
        [PERSON] -
        (NAME) -> [PERSON_NAME]
        (NAME) -> [PERSON_NAME].
    }
END

$S Message From Assertion Subsystem...
$S The asserted graph is unknown to the system and does not contradict it;
$S therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 2.6000000
user: 0.4000000
sys: 0.1300000
child user: 0.0000000
child sys: 0.0000000

(104a) DISPLAY GRAPH END

{
    [PERSON : *2064] -
    (NAME) -> [PERSON_NAME : *2065]
    (NAME) -> [PERSON_NAME : *2066]
}

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

(105) ASSERT GRAPH
    {
        [FAMILY] -
SS Message From Assertion Subsystem...
SS The asserted graph is unknown to the system and does not contradict it;
SS therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 5.4300000
user: 0.3400000
sys: 0.1500000
child user: 0.0000000
child sys: 0.0000000

$5 What Next?
(105a) DISPLAY GRAPH END
{
  [PERSON : *2064] -
  (NAME)->[PERSON_NAME : *2065]
  (NAME)->[PERSON_NAME : *2066]
}
{
  [FAMILY : *2176] -
  (NAME)->[FAMILY_NAME : *2177]
  (NAME)->[FAMILY_NAME : *2178]
}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$5 What Next?
(106) ASSERT GRAPH
{
  [PERSON] -
  (MEMB)<-[FAMILY]
  (MEMB)<-[FAMILY].
}

$5 Message From Assertion Subsystem...
$5 The asserted graph is unknown to the system and does not contradict it;
$5 therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 1.7200000
user: 0.3500000
sys: 0.1000000
child user: 0.0000000
child sys: 0.0000000

$5 What Next?
(106a) DISPLAY GRAPH END
{
  [PERSON : *2064] -
  (NAME)->[PERSON_NAME : *2065]
  (NAME)->[PERSON_NAME : *2066]
}
{
  [FAMILY : *2176] -
  (NAME)->[FAMILY_NAME : *2177]
  (NAME)->[FAMILY_NAME : *2178]
}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(107) ASSERT GRAPH

\{ 
  \[PERSON\] -
  [PERSON] -> [MALENNESS]
  (GENDER) -> [MALENNESS].
\}
END

$\$ Message From Assertion Subsystem...
$\$ The asserted graph is unknown to the system and does not contradict it; 
$\$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 1.6100000
user: 0.3700000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(107a) DISPLAY GRAPH END

\{ 
  \[PERSON\] -
  [PERSON_NAME] : *2065
  (NAME) -> [PERSON_NAME] : *2066
\}

\{ 
  \[FAMILY\] -
  [FAMILY_NAME] : *2178
  (NAME) -> [FAMILY_NAME] : *2177
\}

\{ 
  \[FAMILY\] -
  [FAMILY] : *2205
  (MEMB) -> [PERSON] : *2204
  (MEMB) -> [PERSON] : *2206.
\}

\{ 
  \[PERSON\] : *2232
  (GENDER) -> [MALENNESS] : *2233
  (GENDER) -> [MALENNESS] : *2234
\}

Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(108) ASSERT LATTICE

\{ 
  [MALENNESS] -
  (GENDER) <- [PERSON]
  (GENDER) <- [PERSON].
\}
END
Message From Assertion Subsystem...

The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Time taken (seconds) to perform last macro:

real: 1.6300000
user: 0.4300000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000

What Next?

108a DISPLAY GRAPH END

[PERSON : *2064] -
(NAME)->[PERSON_NAME : *2065]
(NAME)->[PERSON_NAME : *2066]

[FAMILY : *2176] -
(NAME)->[FAMILY_NAME : *2178]
(NAME)->[FAMILY_NAME : *2177]

[FAMILY : *2205] -
(MEMB)->[PERSON : *2204] -
(MEMB)<-[FAMILY : *2206].

[PERSON : *2232] -
(GENDER)->[MALENESS : *2233]
(GENDER)->[MALENESS : *2234]

[MALENESS : *2260] -
(GENDER)<-[PERSON : *2261]
(GENDER)<-[PERSON : *2262]

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0000000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

What Next?

109 ASSERT GRAPH

[PERSON] -
(GENDER)->[FEMALENESS]
(GENDER)->[FEMALENESS].

END

Message From Assertion Subsystem...

The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Time taken (seconds) to perform last macro:

real: 1.6600000
user: 0.4500000
sys: 0.0700000
child user: 0.0000000
child sys: 0.0000000

What Next?

109a DISPLAY GRAPH END

{ }
[PERSON: *2064] -
  (NAME) - [PERSON_NAME: *2065]
  (NAME) - [PERSON_NAME: *2066]
)

{[FAMILY: *2176] -
  (NAME) - [FAMILY_NAME: *2177]
  (NAME) - [FAMILY_NAME: *2178]
}

{[FAMILY: *2206] -
  (MEMB) - [PERSON: *2204] -
  (MEMB) - [FAMILY: *2205],
}

{[PERSON: *2232] -
  (GENDER) - [MALENESS: *2234]
  (GENDER) - [MALENESS: *2233]
}

{[PERSON: *2261] -
  (GENDER) - [MALENESS: *2260]
  (GENDER) - [MALENESS: *2262],
}

{[PERSON: *2288] -
  (GENDER) - [FEMALENESS: *2289]
  (GENDER) - [FEMALENESS: *2290]
}

Time taken (seconds) to perform last macro:

real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$5 What Next?

(110) ASSERT GRAPH
{ [FEMALENESS] -
  (GENDER) - [PERSON] -
  (GENDER) - [PERSON],
}
$5 Message From Assertion Subsystem...
$5 The asserted graph is unknown to the system and does not contradict it;
$5 therefore, this graph is now being stored

Time taken (seconds) to perform last macro:

real: 1.7000000
user: 0.4960000
sys: 0.1100000
child user: 0.0000000
child sys: 0.0000000

$5 What Next?

(110a) DISPLAY GRAPH END

{[PERSON: *2064] -
  (NAME) - [PERSON_NAME: *2065]
  (NAME) - [PERSON_NAME: *2066]
}

{[FAMILY: *2176] -
  (NAME) - [FAMILY_NAME: *2178]
  (NAME) - [FAMILY_NAME: *2177]
}

{[FAMILY: *2205] -
  (MEMB) - [PERSON: *2204] -
(MEMB)<-[FAMILY : *2206].

{[PERSON : *2232] -
 (GENDER)->[MALENESS : *2233]
 (GENDER)->[MALENESS : *2234]
}

{[PERSON : *2262] -
 (GENDER)->[MALENESS : *2260] -
 (GENDER)<-[PERSON : *2261].
}

{[PERSON : *2288] -
 (GENDER)->[FEMALENESS : *2290]
 (GENDER)->[FEMALENESS : *2289]
}

{[FEMALENESS : *2316] -
 (GENDER)<-[PERSON : *2317]
 (GENDER)<-[PERSON : *2318]
}

Time taken (seconds) to perform last macro:

- real: 0.0000000
- user: 0.0000000
- sys: 0.0000000
- child user: 0.0000000
- child sys: 0.0000000

$$ \text{What Next?} $$

(Ill) ASSERT GRAPH

- (ACT) -
  - (RESULT)->[PERSON]
  - (RESULT)->[PERSON].

END

$$ \text{Message From Assertion Subsystem...} $$
$$ \text{The asserted graph is unknown to the system and does not contradict it;} $$
$$ \text{therefore, this graph is now being stored} $$

Time taken (seconds) to perform last macro:

- real: 1.7700000
- user: 0.4200000
- sys: 0.0700000
- child user: 0.0000000
- child sys: 0.0000000

$$ \text{What Next?} $$

(Ill/a) DISPLAY GRAPH END

{[PERSON : *2064] -
 (NAME)->[PERSON_NAME : *2065]
 (NAME)->[PERSON_NAME : *2066]
}

{[FAMILY : *2176] -
 (NAME)->[FAMILY_NAME : *2177]
 (NAME)->[FAMILY_NAME : *2178]
}

{[FAMILY : *2206] -
 (MEMB)->[PERSON : *2204] -
 (MEMB)<-[FAMILY : *2205].
}

{[PERSON : *2232] -
 (GENDER)->[MALENESS : *2234]
 (GENDER)->[MALENESS : *2233]
Time taken (seconds) to perform last macro:
real: 0.0000000
user: 0.0000000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(112) ASSERT GRAPH

( [PERSON] -
(RESULT)<-[ACT]
(RESULT)<-[ACT],

END

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 1.7500000
user: 0.4600000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?

(112a) DISPLAY GRAPH END

( [PERSON : *2064] -
(NAME)<-[PERSON_NAME : *2065]
(NAME)<-[PERSON_NAME : *2066]

)
[PERSON : *2261] -
  (GENDER) -> [MALENESS : *2260] -
  (GENDER) <- [PERSON : *2262] .

[PERSON : *2288] -
  (GENDER) -> [MALENESS : *2290] -
  (GENDER) <- [PERSON : *2282] .

[PERSON : *2318] -
  (GENDER) -> [MALENESS : *2316] -
  (GENDER) <- [PERSON : *2317] .

[ACT : *2344] -
  (RESULT) -> [PERSON : *2345] -
  (RESULT) <- [PERSON : *2346] .

[PERSON : *2372] -
  (RESULT) <- [ACT : *2373] -
  (RESULT) <- [ACT : *2374] .

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(113) ASSESS GRAPH
{
  [PERSON: *x]
  {
    [PERSON: *x] -> (NAME) -> [PERSON_NAME]
  }
}

END

$\$ Message From Assertion Subsystem...
$\$ The asserted graph is unknown to the system and does not contradict it;
$\$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 2.3100000
user: 0.3900000
sys: 0.0700000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(113a) DISPLAY GRAPH END
{
  [PERSON : *2064] -
    (NAME) -> [PERSON_NAME : *2066] -
    (NAME) <- [PERSON : *2065] .

  [FAMILY : *2175] -
    (NAME) <- [FAMILY_NAME : *2178] -
    (NAME) <- [FAMILY : *2177] .

  [FAMILY : *2206] -
    (MEMB) <- [PERSON : *2204] -
    (MEMB) <- [FAMILY : *2205] .
}


[PERSON : *2232] -
  (GENDER)->[MALENESS : *2233]
  (GENDER)->[MALENESS : *2234]
}

{[PERSON : *2261] -
  (GENDER)->[MALENESS : *2260] -
  (GENDER)<-[PERSON : *2262].
}

{[PERSON : *2288] -
  (GENDER)->[FEMALENESS : *2290]
  (GENDER)->[FEMALENESS : *2289]
}

{[PERSON : *2318] -
  (GENDER)->[FEMALENESS : *2316] -
  (GENDER)<<[PERSON : *2317].
}

{[ACT : *2344] -
  (RESULT)->[PERSON : *2346]
  (RESULT)->[PERSON : *2345]
}

{[ACT : *2374] -
  (RESULT)->[PERSON : *2372] -
  (RESULT)<-[ACT : *2373].
}

{[PERSON : *2448]

{[PERSON : *2448] -
  (NAME)->[PERSON_NAME : *2450]
}
)

Time taken (seconds) to perform last macro:
real: 0.0100000
user: 0.0100000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(114) ASSERT GRAPH
{[PERSON_NAME : *x]
  {[PERSON]->(NAME)->[PERSON_NAME : *x]
}
)

$\$ Message From Assertion Subsystem...
$\$ The asserted graph is unknown to the system and does not contradict it;
$\$ therefore, this graph is now being stored.

Time taken (seconds) to perform last macro:
real: 1.7900000
user: 0.3400000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(114a) DISPLAY GRAPH END
{[PERSON : *2064] -
  (NAME)->[PERSON_NAME : *2065]
  (NAME)->[PERSON_NAME : *2066]
(115) ASSERT GRAPH

( [FAMILY : *x] )

( [FAMILY : *x] -> [MEMB] -> [PERSON] )
SS Message From Assertion Subsystem...
SS The asserted graph is unknown to the system and does not contradict it; therefore this graph is now being stored

Time taken (seconds) to perform last macro:
real: 1.6200000
user: 0.3600000
sys: 0.0500000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(115a) DISPLAY GRAPH END

{ [PERSON : *2064] -
  (NAME)->[PERSON_NAME : *2065]
  (NAME)->[PERSON_NAME : *2066]
}

{ [FAMILY : *2176] -
  (NAME)->[FAMILY_NAME : *2177]
  (NAME)->[FAMILY_NAME : *2178]
}

{ [FAMILY : *2206] -
  (MEMB)->[PERSON : *2204] -
  (MEMB)<-[FAMILY : *2205],
}

{ [PERSON : *2232] -
  (GENDER)->[MALENESS : *2234]
  (GENDER)->[MALENESS : *2233]
}

{ [PERSON : *2262] -
  (GENDER)->[MALENESS : *2260] -
  (GENDER)<-[PERSON : *2261],
}

{ [PERSON : *2288] -
  (GENDER)->[FEMALENESS : *2289]
  (GENDER)->[FEMALENESS : *2290]
}

{ [PERSON : *2317] -
  (GENDER)->[FEMALENESS : *2316] -
  (GENDER)<-[PERSON : *2318],
}

{ [ACT : *2344] -
  (RESULT)->[PERSON : *2346]
  (RESULT)->[PERSON : *2345]
}

{ [ACT : *2373] -
  (RESULT)->[PERSON : *2372] -
  (RESULT)<-[ACT : *2374],
}

{ [PERSON : *2448] }
(NAME)->[PERSON_NAME : *2472]

(FAMILY : *2496)

(FAMILY : *2496) -
(MEMB)->[PERSON : *2498]

Time taken (seconds) to perform last macro:

real: 0.02000000
user: 0.01000000
sys: 0.01000000
child user: 0.00000000
child sys: 0.00000000

What Next?

(116) ASSERT GRAPH

{ [PERSON: *x]

( [ACT]->(RESULT)->[PERSON: *x]

) END

(Message From Assertion Subsystem...

The asserted graph is unknown to the system and does not contradict it;

therefore, this graph is now being stored

Time taken (seconds) to perform last macro:

real: 1.73000000
user: 0.41000000
sys: 0.11000000
child user: 0.00000000
child sys: 0.00000000

What Next?

(116a) DISPLAY GRAPH END

{ [PERSON : *2064] -

(NAME)->[PERSON_NAME : *2066] (NAME)->[PERSON_NAME : *2065]

} (FAMILY : *2176) -

(NAME)->[FAMILY_NAME : *2178] (NAME)->[FAMILY_NAME : *2177]

} (FAMILY : *2205) -

(MEMB)->[PERSON : *2204] - (MEMB)<-[FAMILY : *2206].

} [PERSON : *2232] -

(GENDER)->[MALENESS : *2234] (GENDER)->[MALENESS : *2233]

} [PERSON : *2261] -

(GENDER)->[MALENESS : *2260] - (GENDER)<-[PERSON : *2262].

} [PERSON : *2288] -

(GENDER)->[FEMALENESS : *2290] (GENDER)->[FEMALENESS : *2289]
Time taken (seconds) to perform last macro:

real: 0.0200000
user: 0.0200000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$$\textit{What Next?}$$

(117) \textbf{ASSERT GRAPH}

{ [ACT: \text{*x}] 
  [ACT: \text{*x}\rightarrow(RESULT)\rightarrow\text{[PERSON]}}

$$\text{END}$$

$$\text{END}$$

$$\text{END}$$

$\textit{Message from Assertion Subsystem...}$

$\text{The asserted graph is unknown to the system and does not contradict it;}$

$\text{therefore, this graph is now being stored}$

Time taken (seconds) to perform last macro:

real: 1.7100000
user: 0.3100000
sys: 0.1100000
child user: 0.0000000
child sys: 0.000000

$$\text{What Next?}$$

(117a) DISPLAY GRAPH END

(PERSON *2064) -
(NAME)->(PERSON_NAME : *2066)
(NAME)->(PERSON_NAME : *2065)

(FAMILY *2176) -
(NAME)->(FAMILY_NAME : *2177)
(NAME)->(FAMILY_NAME : *2178)

(FAMILY *2206) -
(MEMBER)->(PERSON : *2204) -
(MEMBER)<->[FAMILY : *2205],

(PERSON *2232) -
(GENDER)->[MALENESS : *2233]
(GENDER)->[MALENESS : *2234]

(PERSON *2261) -
(GENDER)->[MALENESS : *2260] -
(GENDER)<->[PERSON : *2261],

(PERSON *2288) -
(GENDER)->[FEMALENESS : *2289]
(GENDER)->[FEMALENESS : *2290]

(PERSON *2317) -
(GENDER)->[FEMALENESS : *2316] -
(GENDER)<->[PERSON : *2318],

(ACT *2344) -
RESULT)->(PERSON : *2345)
RESULT)->(PERSON : *2346)

(ACT *2374) -
RESULT)->(PERSON : *2372) -
RESULT)<->[ACT : *2373],

(PERSON *2448)

(PERSON *2448) -
(NAME)->[PERSON_NAME : *2450]

(PERSON_NAME : *2472)

(PERSON *2473) -
(NAME)->[PERSON_NAME : *2472]

(FAMILY *2496)

(FAMILY *2496) -
(MEMBER)->[PERSON : *2498]
(PERSON : *2520)

[ACT : *2521] -
  (RESULT) -> [PERSON : *2520]
)
)

[ACT : *2544]

[ACT : *2544] -
  (RESULT) -> [PERSON : *2546]
)
)

Time taken (seconds) to perform last macro:
  real: 0.0200000
  user: 0.0200000
  sys: 0.0000000
  child user: 0.0000000
  child sys: 0.0000000

$ What Next?

(116) ASSERT GRAPH
{
  (PERSON: *x)
  
  (FAMILY) - [MEMB] - (PERSON: *x)
}
END

$ Message From Assertion Subsystem...
$ The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
  real: 2.6100000
  user: 0.3300000
  sys: 0.1100000
  child user: 0.0000000
  child sys: 0.0000000

$ What Next?

(116a) DISPLAY GRAPH END
{
  (PERSON: *2064)
  (NAME) -> [PERSON_NAME: *2065]
  (NAME) -> [PERSON_NAME: *2066]
}

(FAMILY: *2176)
  (NAME) -> [FAMILY_NAME: *2178]
  (NAME) -> [FAMILY_NAME: *2177]

(FAMILY: *2205)
  (MEMB) -> [PERSON: *2204] -
  (MEMB) <- [FAMILY: *2205],

(PERSON: *2232)
  (GENDER) -> [MALENESS: *2233]
  (GENDER) -> [MALENESS: *2234]

(PERSON: *2261)
  (GENDER) -> [MALENESS: *2260] -
  (GENDER) <- [PERSON: *2262],
Time taken (seconds) to perform last macro:

real: 0.0300000
user: 0.0300000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000
assert graph
{
    [maleness: *x]
    {
        [person] -> [gender] -> [maleness: *x]
    }
}
end

message from assertion subsystem...
the asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

time taken (seconds) to perform last macro:
real: 1.7300000
user: 0.3600000
sys: 0.1500000
child user: 0.0000000
child sys: 0.0000000

display graph end

([person : 2064] -
    [name] -> [person_name : 2066]
    [name] -> [person_name : 2065]
)

([family : 2176] -
    [name] -> [family_name : 2177]
    [name] -> [family_name : 2178]
)

([family : 2205] -
    [mem] -> [person : 2204] -
    [mem] <- [family : 2206],
)

([person : 2232] -
    [gender] -> [maleness: 2234]
    [gender] -> [maleness: 2233]
)

([person : 2262] -
    [gender] -> [maleness: *2260] -
    [gender] <- [person: 2261],
)

([person : 2288] -
    [gender] -> [femaleness: *2289]
    [gender] -> [femaleness: *2290]
)

([person : 2318] -
    [gender] -> [femaleness: *2316] -
    [gender] <- [person: *2317],
)

([act : 2344] -
    [result] -> [person: 2345]
    [result] -> [person: *2346]
)

([act : 2374] -
    [result] -> [person: 2372] -
    [result] <- [act: 2371],
)

([person : 2448] -
[PERSON : *2448] -
  (NAME)->[PERSON_NAME : *2450]
)
)

[PERSON_NAME : *2472]

[PERSON : *2473] -
  (NAME)->[PERSON_NAME : *2472]
)
)

[FAMILY : *2496]

[FAMILY : *2496] -
  (MKB)->[PERSON : *2498]
)
)

[PERSON : *2520]

[ACT : *2521] -
  (RESULT)->[PERSON : *2520]
)
)

[ACT : *2544]

[ACT : *2544] -
  (RESULT)->[PERSON : *2546]
)
)

[PERSON : *2568]

[FAMILY : *2569] -
  (MKB)->[PERSON : *2568]
)
)

[MALENESS : *2592]

[PERSON : *2593] -
  (GENDER)->[MALENESS : *2592]
)
)

Time taken (seconds) to perform last macro:
real: 0.0300000
user: 0.0200000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?
(120) assert graph
  [FAMILY: *x]
  
  (FAMILY: *x)->(NAME)->[FAMILY_NAME]

END

$\$ Message From Assertion Subsystem...
$\$ The asserted graph is unknown to the system and does not contradict it;
$\$ therefore, this graph is now being stored.

Time taken (seconds) to perform last macro:
What Next?

(120a) DISPLAY GRAPH END

{[PERSON : *2064] -
 (NAME)->[PERSON_NAME : *2066]
 (NAME)->[PERSON_NAME : *2065]
}

{[FAMILY : *2176] -
 (NAME)->[FAMILY_NAME : *2178]
 (NAME)->[FAMILY_NAME : *2177]
}

{[FAMILY : *2206] -
 (MEMB)->[PERSON : *2204] -
 (MEMB)<-[FAMILY : *2205].
}

{[PERSON : *2232] -
 (GENDER)->[MALENESS : *2233]
 (GENDER)->[MALENESS : *2234]
}

{[PERSON : *2261] -
 (GENDER)->[MALENESS : *2260] -
 (GENDER)<-[PERSON : *2262],
}

{[PERSON : *2280] -
 (GENDER)->[FEMALENESS : *2290]
 (GENDER)->[FEMALENESS : *2292]
}

{[PERSON : *2317] -
 (GENDER)->[FEMALENESS : *2316] -
 (GENDER)<-[PERSON : *2318],
}

{[ACT : *2344] -
 (RESULT)->[PERSON : *2346]
 (RESULT)->[PERSON : *2345]
}

{[ACT : *2373] -
 (RESULT)->[PERSON : *2372] -
 (RESULT)<-[ACT : *2374].
}

{[PERSON : *2448] -
 [PERSON : *2448] -
 (NAME)->[PERSON_NAME : *2450]
}

{[PERSON_NAME : *2472] -
 [PERSON : *2473] -
 (NAME)->[PERSON_NAME : *2472]
}

{[FAMILY : *2496] -
 [FAMILY : *2496] -
(MEMBER) -> (PERSON: *2498)

{ (PERSON: *2520)
  (ACT: *2521) -> (RESULT) -> (PERSON: *2520)
}

{ (ACT: *2544)
  (RESULT) -> (PERSON: *2546)
}

{ (PERSON: *2568)
  (FAMILY: *2569) -> (MEMBER) -> (PERSON: *2568)
}

{ (MALENESS: *2592)
  (PERSON: *2591) -> (GENDER) -> (MALENESS: *2592)
}

{ (FAMILY: *2616)
  (NAME) -> (FAMILY_NAME: *2618)
}

Time taken (seconds) to perform last macro:
real: 0.0300000
user: 0.0300000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(121) ASSERT GRAPH
  
  [STATE: *x]
  
  { [STATE: *x] -> (EXPR) -> (PERSON) }

END

SS Message From Assertion Subsystem...
SS The asserted graph is unknown to the system and does not contradict it; 
SS therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 1.9000000
user: 0.4100000
sys: 0.0900000
child user: 0.0000000
child sys: 0.0000000

SS What Next?

(121a) DISPLAY GRAPH END
(ACT *2544J

[ACT : *2544] -
  (RESULT) -> [PERSON : *2546]

[PERSON : *2568]

[PERSON : *2593] -
  (GENDER) -> [MALENESS : *2592]

[FAMILY : *2569] -
  (MEMB) -> [PERSON : *2568]

[FAMILY : *2616]

(FAMILY : *2616) -
  (NAME) -> [FAMILY_NAME : *2618]

[STATE : *2640]

[STATE : *2640] -
  (EXPR) -> [PERSON : *2642]

Time taken (seconds) to perform last macro:
  real: 0.0300000
  user: 0.0300000
  sys: 0.0000000
child user: 0.0000000
  child sys: 0.0000000

$$ What Next?

(122) ASSERT GRAPH
  [FAMILY_NAME: *x]
    (FAMILY) -> (NAME) -> [FAMILY_NAME: *x]

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
  real: 2.0500000
  user: 0.4000000
  sys: 0.1300000
child user: 0.0000000
  child sys: 0.0000000

$$ What Next?

(122a) DISPLAY GRAPH END
(ACT *2544)

  (ACT *2544) -
    (RESULT)->(PERSON *2546)

) )

  (PERSON *2568)

  (FAMILY *2569) -
    (MEMB)->(PERSON *2568)

) )

  (MALENESS *2592)

  (PERSON *2593) -
    (GENDER)->(MALENESS *2592)

) )

  (FAMILY *2616)

  (FAMILY *2616) -
    (NAME)->(FAMILY_NAME *2618)

) )

  (STATE *2640)

  (STATE *2640) -
    (EXPR)->(PERSON *2642)

) )

  (FAMILY_NAME *2688)

  (FAMILY *2689) -
    (NAME)->(FAMILY_NAME *2688)

) )

Time taken (seconds) to perform last macro:

real: 0.0300000
user: 0.0300000
sys: 0.0000000
child user: 0.0000000
child sys: 0.0000000

$ What Next?

(123) ASSERT GRAPH

  (MALENESS *x)

    (PERSON)->(GENDER)->(MALENESS *x)

) END

$ Message From Assertion Subsystem...

$ The asserted graph is unknown to the system and does not contradict it;
$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:

real: 1.9600000
user: 0.5100000
sys: 0.1200000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(233a) DISPLAY GRAPH END

```plaintext
[PERSON : *2064] -
  (NAME) -> [PERSON_NAME : *2066]
  (NAME) -> [PERSON_NAME : *2065]  

[FAMILY : *2176] -
  (NAME) -> [FAMILY_NAME : *2176]
  (NAME) -> [FAMILY_NAME : *2177]  

[FAMILY : *2206] -
  (MEMB) -> [PERSON : *2204] -
  (MEMB) <- [FAMILY : *2205]  

[PERSON : *2232] -
  (GENDER) -> [MALENESS : *2233]
  (GENDER) -> [MALENESS : *2234]  

[PERSON : *2261] -
  (GENDER) -> [MALENESS : *2260] -
  (GENDER) <- [PERSON : *2262]  

[PERSON : *2288] -
  (GENDER) -> [FEMALENESS : *2290]
  (GENDER) -> [FEMALENESS : *2289]  

[PERSON : *2318] -
  (GENDER) -> [FEMALENESS : *2316] -
  (GENDER) <- [PERSON : *2317]  

[ACT : *2344] -
  (RESULT) -> [PERSON : *2346]
  (RESULT) -> [PERSON : *2345]  

[ACT : *2373] -
  (RESULT) -> [PERSON : *2372] -
  (RESULT) <- [ACT : *2374]  

[PERSON : *2448]  

[PERSON : *2448] -
  (NAME) -> [PERSON_NAME : *2450]  

[PERSON_NAME : *2472]  

[PERSON : *2473] -
  (NAME) -> [PERSON_NAME : *2472]  

[FAMILY : *2496]  

[FAMILY : *2496] -
  (MEMB) -> [PERSON : *2498]  
```
Time taken (seconds) to perform last macro:
real: 0.0300000
user: 0.0200000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$$ What Next?$$
-- [PERSON: *2520]

-- {ACT: *2521] -> (RESULT) -> [PERSON: *2520]

-- [ACT: *2544]

-- {ACT: *2544] -> (RESULT) -> [PERSON: *2546]

-- [PERSON: *2568]

-- {FAMILY: *2568] -> (MEMB) -> [PERSON: *2568]

-- [MALENES: *2592]

-- [PERSON: *2593] -> (GENDER) -> [MALENES: *2592]

-- [FAMILY: *2616]

-- {FAMILY: *2616] -> [FAMILY_NAME: *2618]

-- [STATE: *2640]

-- {STATE: *2640] -> [PERSON: *2642]

-- [FAMILY_NAME: *2688]

-- {FAMILY: *2689] -> [FAMILY_NAME: *2688]

-- [FEMALENESS: *2712]

-- [PERSON: *2713] -> (GENDER) -> [FEMALENESS: *2712]

Time taken (seconds) to perform last macro:
real: 0.0300000
user: 0.0200000
sys: 0.0100000
child user: 0.0000000
child sys: 0.0000000

$ What Next?
(124) ASSERT GRAPH
    [PERSON] -> [PERSON_NAME: "Brian"]
(MEMB)<-(FAMILY) -
(NAME)->(FAMILY_NAME: "Bowen")..

END

$\$ Message From Assertion Subsystem...
$\$ The asserted graph is unknown to the system and does not contradict it;
$\$ therefore, this graph is now being stored

Time taken (seconds) to perform last macro:
real: 3.2700000
user: 0.5900000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000

$\$ What Next?

(124a) DISPLAY GRAPH END

[FAMILY : *2737] -
(NAME)->(FAMILY_NAME : *2751)
(MEMB)->[PERSON : *2736] -
(NAME)->(PERSON_NAME : *2750),

{[PERSON : *2064] -
(NAME)->(PERSON_NAME : *2066)
(NAME)->(PERSON_NAME : *2065)}

{[FAMILY : *2176] -
(NAME)->(FAMILY_NAME : *2178)
(NAME)->(FAMILY_NAME : *2177)}

{[FAMILY : *2206] -
(MEMB)->[PERSON : *2204] -
(MEMB)<-(FAMILY : *2205),

{[PERSON : *2232] -
(GENDER)->[MALENESS : *2233]
(GENDER)->[MALENESS : *2234]}

{[PERSON : *2261] -
(GENDER)->[MALENESS : *2260] -
(GENDER)<-(PERSON : *2262),

{[PERSON : *2288] -
(GENDER)->[FEMALENESS : *2290]
(GENDER)->[FEMALENESS : *2289]}

{[PERSON : *2318] -
(GENDER)->[FEMALENESS : *2316] -
(GENDER)<-(PERSON : *2317),

{[ACT : *2344] -
<Result)->[PERSON : *2346]
<Result)->[PERSON : *2345]}

{[ACT : *2373] -
<Result)->[PERSON : *2372] -
(Result)<-(ACT : *2374),

{[PERSON : *2448] -

{[PERSON : *2448] -
(NAME)->(PERSON_NAME : *2450]
Time taken (seconds) to perform last macro:

real: 0.0300000
user: 0.0100000
sys: 0.0200000
child user: 0.0000000
child sys: 0.0000000
Testing of Assertion and Inquiry

We here present some figures (not an evaluation) concerning the operation and timing of the assertion and inquiry macro operations. The first test of GGEN.PC (which follows these figures) produces the following output for a maximum graph size of five dyads, and for the production of five graphs:

Maximum Number of Dyads in Test Graph?
{PERSON : *6875} -
  (GENDER) -> [MALENESS : *6876] -
  (GENDER) <= {PERSON : *6877} -
  (GENDER) -> [FEMALENESS : *6879],
  (GENDER) <= {PERSON : *6878},

Number of Graphs?
{PERSON : #1} -
  (GENDER) -> [MALENESS : #2] -
  (GENDER) <= {PERSON : #3} -
  (GENDER) -> [FEMALENESS : #5],
  (GENDER) <= {PERSON : #4},
$$\text{SSI}$$ Message From Assertion Subsystem...
$$\text{SSI}$$ The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored

Time taken (seconds) to perform that Assertion...

real: 1.7000000
user: 0.4900000
sys: 0.1300000
child user: 0.0000000
child sys: 0.0000000

{PERSON : #6} -
  (GENDER) -> [MALENESS : #7] -
  (GENDER) <= {PERSON : #8} -
  (GENDER) -> [FEMALENESS : #10],
  (GENDER) <= {PERSON : #9},
$$\text{SSI}$$ Message From Assertion Subsystem...
$$\text{SSI}$$ The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored

Time taken (seconds) to perform that Assertion...

real: 1.7400000
user: 0.5000000
sys: 0.1100000
child user: 0.0000000
child sys: 0.0000000

{PERSON : #11} -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <= {PERSON : #13} -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <= {PERSON : #14},
$$\text{SSI}$$ Message From Assertion Subsystem...
$$\text{SSI}$$ The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored

Time taken (seconds) to perform that Assertion...

real: 1.5800000
user: 0.4900000
sys: 0.0500000
child user: 0.0000000
child sys: 0.0000000
[PERSON : #16] -
  (GENDER)->[MALENESS : #17] -
  (GENDER)->[PERSON : #18] -
  (GENDER)->[FEMALENESS : #20],
  (GENDER)<-[PERSON : #19],

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored.

Time taken (seconds) to perform that Assertion:

real: 2.4800000
user: 0.4300000
sys: 0.1200000
child user: 0.0000000
child sys: 0.0000000

[PERSON : #21] -
  (GENDER)->[MALENESS : #22] -
  (GENDER)<-[PERSON : #23] -
  (GENDER)->[FEMALENESS : #25],
  (GENDER)<-[PERSON : #24],

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored.

Time taken (seconds) to perform that Assertion:

real: 2.9500000
user: 0.4600000
sys: 0.0900000
child user: 0.0000000
child sys: 0.0000000

Instr CREATE VIEW V1 (C1_PERSON__6875, C2_MALENESS__6876, C3_PERSON__6877,
C4_FEMALENESS__6879, C5_PERSON__6878) AS SELECT UNIQUE T1.C1, T1.C2, T2.C1, T3.C2,
T4.C1 FROM L147P T1, L147P T2, L148P T3, L147P T4 WHERE T1.C1 IS NOT NULL AND T1.C2 IS
NOT NULL AND T2.C1 IS NOT NULL AND T3.C2 IS NOT NULL AND T4.C1 IS NOT NULL AND
Instr2 SELECT UNIQUE C1_PERSON__6875, C2_MALENESS__6876, C3_PERSON__6877,
C4_FEMALENESS__6879, C5_PERSON__6878 FROM V1

[MAN : #11] -
  (GENDER)->[MALENESS : #2] -
  (GENDER)<-[MAN : #3] -
  (GENDER)->[FEMALENESS : #5],
  (GENDER)<-[MAN : #11],

[MAN : #11] -
  (GENDER)->[MALENESS : #2] -
  (GENDER)<-[MAN : #3] -
  (GENDER)->[FEMALENESS : #5],
  (GENDER)<-[MAN : #3],

[MAN : #11] -
  (GENDER)->[MALENESS : #2] -
  (GENDER)<-[MAN : #3] -
  (GENDER)->[FEMALENESS : #5],
  (GENDER)<-[MAN : #4],

[MAN : #11] -
  (GENDER)->[MALENESS : #12] -
  (GENDER)<-[MAN : #13] -
  (GENDER)->[FEMALENESS : #15],
  (GENDER)<-[MAN : #11],

[MAN : #11] -
  (GENDER)->[MALENESS : #12] -
  (GENDER)<-[MAN : #13] -
  (GENDER)->[FEMALENESS : #15],
  (GENDER)<-[MAN : #13],

[MAN : #11] -
  (GENDER)->[MALENESS : #12] -
  (GENDER)<-[MAN : #13] -
  (GENDER)->[FEMALENESS : #15],
  (GENDER)<-[MAN : #14],
[MAN : #13] -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <- [MAN : #13] -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <- [MAN : #11],

[MAN : #13] -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <- [MAN : #13] -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <- [MAN : #13],

[MAN : #14] -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <- [MAN : #13] -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <- [MAN : #14],

[MAN : #14] -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <- [MAN : #13] -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <- [MAN : #13],

[MAN : #14] -
  (GENDER) -> [MALENESS : #12] -
  (GENDER) <- [MAN : #13] -
  (GENDER) -> [FEMALENESS : #15],
  (GENDER) <- [MAN : #14],

[MAN : #16] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #16] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #16],

[MAN : #16] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #18],

[MAN : #16] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #19],

[MAN : #18] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #16],

[MAN : #18] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #18],

[MAN : #19] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #16],

[MAN : #19] -
  (GENDER) -> [MALENESS : #17] -
  (GENDER) <- [MAN : #18] -
  (GENDER) -> [FEMALENESS : #20],
  (GENDER) <- [MAN : #19],
Time taken (seconds) to perform Inquiry:

real: 19.0800000
user: 18.0000000
sys: 0.1300000
child user: 0.0000000
child sys: 0.0000000

Assertion of such graphs take from 1½ to 3 seconds; retrieving those graphs from secondary storage (which involves the use of \( \phi_A \) and display mechanisms) takes nearly 20 seconds for form a call, retrieve the graphs, and to display them. As the
performance of $\phi_A$ has been described elsewhere in terms of tens of seconds, and
display does not require a great amount of time, the overhead is therefore in the
communication, interpretation, and performance of the issued instruction by Oracle.

The second test of GGEN.PC produces the following output for a maximum graph
size of five dyads, and for the production of one hundred graphs:

```
Maximum Number of Dyads in Test Graph?
[PERSON : *6977] -
  (RELATIVE)->[PERSON : *6978] -
  (NAME)->[PERSON_NAME : *6979] -
  (NAME)<-[PERSON : *6981],
  (FRIEND)->[PERSON : *6980].

Number of Graphs?
[PERSON : #1] -
  (RELATIVE)->[PERSON : #2] -
  (NAME)->[PERSON_NAME : "STR3"] -
  (NAME)<-[PERSON : #5],
  (FRIEND)->[PERSON : #4],

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored

Time taken (seconds) to perform that Assertion:
real: 1.6400000
user: 0.5200000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000

[PERSON : #6] -
  (RELATIVE)->[PERSON : #7] -
  (NAME)->[PERSON_NAME : "STR8"] -
  (NAME)<-[PERSON : #10],
  (FRIEND)->[PERSON : #9],

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored

<Here, nearly 100 assertions have been deleted>

[PERSON : #481] -
  (RELATIVE)->[PERSON : #482] -
  (NAME)->[PERSON_NAME : "STR483"] -
  (NAME)<-[PERSON : #485],
  (FRIEND)->[PERSON : #484],

$$ Message From Assertion Subsystem...
$$ The asserted graph is unknown to the system and does not contradict it;
$$ therefore, this graph is now being stored

Time taken (seconds) to perform that Assertion:
real: 2.8000000
user: 0.4600000
sys: 0.1100000
child user: 0.0000000
child sys: 0.0000000
```
The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Time taken (seconds) to perform that Assertion:

real: 2.1200000
user: 0.4300000
sys: 0.0900000
child user: 0.5000000
child sys: 0.0000000

The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Time taken (seconds) to perform that Assertion:

real: 1.6000000
user: 0.4300000
sys: 0.1000000
child user: 0.0000000
child sys: 0.0000000

The asserted graph is unknown to the system and does not contradict it; therefore, this graph is now being stored.

Time taken (seconds) to perform that Assertion:

real: 1.7500000
user: 0.4400000
sys: 0.0800000
child user: 0.0000000
child sys: 0.0000000


Instr2 SELECT UNIQUE C1_PERSON __ 6977, C2_PERSON __ 6978, C3_PERSON_NAME __ 6979, C4_PERSON__6981, C5_PERSON__6980 FROM V1
<Here, around 200 instantiations have been deleted>

The program upon which these tests are based follows.
EXEC SQL INCLUDE $QLCA;
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/times.h>
#include <unistd.h>
#include <defines.h>
#include "cnclass.h"

static void pr_times (real, tmsstart, tmsend)
    clock_t real;
    struct tms *tmsstart, *tmsend;
{
    static long clktck = 0;
    /***************************************************************************/
    /* Timing Display Routine. */
    /***************************************************************************/
    clktck = sysconf (_SC_CLK_TCK);
    printf(" real: %7.7f\n", (double) clktck);
    printf(" user: %7.7f\n", (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf(" sys: %7.7f\n", (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf(" child user: %7.7f\n", (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf(" child sys: %7.7f\n", (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}

/*****************************************************
/* Look for the END token that MUST end all inputs. */
/*****************************************************/

int EndTest (Buff, StrLen)
    BufferUnit *Buff;
    int StrLen;
{
    if (StrLen < 2) {
        return (FALSE);
    } else {
        if (Buff->Line[StrLen-2] == 'E') {
            if (Buff->Line[StrLen-1] == 'N') {
                if (Buff->Line[StrLen] == 'D') {
                    return (TRUE);
                } else {
                    return (FALSE);
                }
            } else {
                return (FALSE);
            }
        } else {
            if (Buff->Line[StrLen-3] == 'Q'
                && Buff->Line[StrLen-2] == 'U'
                && Buff->Line[StrLen-1] == 'I'
                && Buff->Line[StrLen] == 'T') {
                return (TRUE);
            } else {
                return (FALSE);
            }
        }
    }

char *TraverseLattice (Latt, Num)
    LatticeElt *Latt;
    int Num;
{
    typedef struct {
        LatticeElt *NodeAddr;
        int Traversed;
        struct NodesDone *NextInList;
    } NodesDone;
    NodesDone *TopNode, *NewNode, *EndNode, *ScanNodesDone;
    LatticeElt *CurrNode, *FoundNode;
    LatticeEltPtr *Subs;
    int CountOfNodes;
TopNode = typealloc (NodeSDone);
TopNode->NodeAddr = Latt;
TopNode->Traversed = FALSE;
TopNode->NextlnList = 0;
CountOfNodes = 1;

for (ScanNodesDone = TopNode; CountOfNodes < Num && ScanNodesDone != 0; ScanNodesDone = ScanNodesDone->NextlnList)
    if (ScanNodesDone->Traversed == FALSE){
        ScanNodesDone->Traversed = TRUE;
        CurrNode = ScanNodesDone->NodeAddr;
        for (Sub = CurrNode->SubLabels; CountOfNodes < Num && Sub != 0; Sub = Sub->NextlnList){
            FoundNode = Sub->PointedAt;
            NewNode = typealloc (NodesDone);
            NewNode->NodeAddr = FoundNode;
            NewNode->Traversed = FALSE;
            NewNode->NextlnList = 0;
            for (EndNode = TopNode; EndNode->NextlnList != 0; EndNode = EndNode->NextlnList){
                if (EndNode == TopNode; EndNode->NextlnList != 0; EndNode = EndNode->NextlnList)
                    CountOfNodes++;
            }
        }
    } else {
        CountOfNodes++;
    }
    for (ScanNodesDone = TopNode; ScanNodesDone->NextlnList != 0; ScanNodesDone = ScanNodesDone->NextlnList)
        CurrNode = ScanNodesDone->NodeAddr;
        return (CurrNode->Label);
}

void NodeCount (Node, Count)
LatticeEl *Node;
int *Count;
{ LatticeElPtr *FoundNodePtr;
LatticeEl *FoundNode;
for (FoundNodePtr = Node->SubLabels; FoundNodePtr != 0; FoundNodePtr = FoundNodePtr->NextlnList){
    (*Count)++;
}
}

void MakeData (Graph, NumOfG, TOL, RDL, FDL, TL, RL, Cards, SOG, High)
ContextListEl *Graph, *SOG;
int NumOfG, *High;
Defn *TOL, *RDL, *FDL;
LatticeEl *TL, *RL;
Cardinality *Cards;

struct tms tmsstart, tmsend;
clock_t start, end;
Cardinality **C;
ContextListEl *NewGraph;
Context *Cxt, *C, *S;
ConceptListEl *CLELoop;
Concept *ConcLoop;
int Referents, Counter;
char *Quote = ", ; *QuoteS = *\STR", *Hash = *\", *StrIndv = *\INDIVIDUAL\0", *StrString = *STRING\0";

Referents = 1;
for (Counter = 1; Counter <= NumOfG; Counter++){
    NewGraph = ShadowCopy (Graph, 0);
    Cxt = NewGraph->PointedAt;
    for (CLELoop = Cxt->ListOfMembers; CLELoop != 0; CLELoop = CLELoop->NextlnList){
        strncpy (ConcLoop->Referent, '\0', 25);
        if (Sub (StrIndv, ConcLoop->Label, TL, FALSE, ANY) == TRUE){
            strncpy (ConcLoop->Referent, Hash, 1);
            strcat (ConcLoop->Referent, IntToString (Referents));
            References++;
        } else {
            if (Sub (StrString, ConcLoop->Label, TL, FALSE, ANY) == TRUE){
                strncpy (ConcLoop->Referent, QuoteS, 4);
                strcat (ConcLoop->Referent, IntToString (References));
                References++;
            } else {

```
```c
strcat (ConcLoop->Referent, IntToString (Referents));
Referents++;
}
}
G = &Graph;
S = &SOO;
C = &Cards;
Graph_Display_Gateway (GRAPH, '\0', TDL, RDL, FDL, NewGraph, SOO, TL, RL);
start = times (&tmsstart);
Assertion (GRAPH, '\0', NewGraph, '\0', 'TDL', 'RDL', 'FDL', G, S, TL, RL, High, C);
end = times (&tmsend);
printf ("\nTime taken (seconds) to perform that Assertion...

-\n");
pr_times (end-start, &tmsstart, &tmsend);

void GenSimpleGraph (TLattice, TLSize, RLattice, RLSize, NumOfDyads, Cards, TDL, RDL, FDL, SOO, High)
LatticeElt *TLattice, *RLattice;
int TLSize, RLSize, NumOfDyads;
Defn *TDL, *RDL, *FDL;
Cardinality *Cards;
ContextListElt *SOO;

type def struct {
    Concept *ConcAddr;
    struct StackElt *NextInList;
} StackElt;

struct tms tmssstart, tmsend;
clock_t start, end;
Cardinality *ScanCards;
ContextListElt *Complex, *Copy;
Context *Complex_Cxt;
ConceptListElt *Complex_SimpleGraphPtr;
Relation *R;
RelationListElt *NullRL;
char KL[25], K2Lab[25], RLab[25], CLab[25], Referent[25], *Var = "-",
int NumOfCards, RandCard, FindCards, RandNum, SLen, NMinusOne, DyadsInGraph, n, ScanBase, GetOut,
MaxGraphSize, NumOfGraphs;

char *Punct;
int InitLoop;
ConceptListElt *StackOfConcs;

/********************************************************/

void GenSimpleGraph (TLattice, TLSize, RLattice, RLSize, NumOfDyads, Cards, TDL, RDL, FDL, SOO, High)
LatticeElt *TLattice, *RLattice;
int TLSize, RLSize, NumOfDyads;
Defn *TDL, *RDL, *FDL;
Cardinality *Cards;
ContextListElt *SOO;

type def struct {
    Concept *ConcAddr;
    struct StackElt *NextInList;
} StackElt;

struct tms tmssstart, tmsend;
clock_t start, end;
Cardinality *ScanCards;
ContextListElt *Complex, *Copy;
Context *Complex_Cxt;
ConceptListElt *Complex_SimpleGraphPtr;
Relation *R;
RelationListElt *NullRL;
char KL[25], K2Lab[25], RLab[25], CLab[25], Referent[25], *Var = "-",
int NumOfCards, RandCard, FindCards, RandNum, SLen, NMinusOne, DyadsInGraph, n, ScanBase, GetOut,
MaxGraphSize, NumOfGraphs;

char *Punct;
int InitLoop;
ConceptListElt *StackOfConcs;

/********************************************************/

for (ScanCards = Cards; ScanCards != 0; ScanCards = ScanCards->NextInList) {
    NumOfCards++;
    RandCard = (rand() % NumOfCards) + 1;
    FindCards = 1;
    GetOut = FALSE;
```
for (ScanCards = Cards; GetOut == FALSE && ScanCards != 0; ScanCards = ScanCards->NextInList){
    if (FindCards == RandCard){
        GetOut = TRUE;
        strncpy (K1Lab, '\0', 25);
        strncpy (K1Lab, ScanCards->FromPos, LongestString (ScanCards->FromPos, '\0'));
        strncpy (K2Lab, '\0', 25);
        strncpy (K2Lab, ScanCards->ToPos, LongestString (ScanCards->ToPos, '\0'));
        strncpy (RLab, '\0', 25);
        strncpy (RLab, ScanCards->Relation, LongestString (ScanCards->Relation, '\0'));
    } else {
        FindCards++;
    }
}

K1 = typealloc (Concept);
K2 = typealloc (Concept);
R = typealloc (Relation);
strncpy (Referent, '\0', 25);
strncpy (Referent, ScanCards->FromPos, LongestString (ScanCards->FromPos, '\0'));
strncpy (Referent, ScanCards->ToPos, LongestString (ScanCards->ToPos, '\0'));
strncpy (Referent, ScanCards->Relation, LongestString (ScanCards->Relation, '\0'));
}

// Now generate dyads to make the graph up to NumOfDyads size. We saved both */
// of the concepts just generated in a stack, and we call one concept from /*
// stack (randomly) in each dyad creation - this ensures connectivity of the /*
// graph generated. Each new concept is also added to the stack, so that it */
// may be used later during a later iteration.

SLEN = 2;
NMinusOne = NumOfDyads--;
MaxGraphSize = rand() % NMinusOne;
for (DyadsInGraph = 1; DyadsInGraph <= MaxGraphSize; DyadsInGraph++){
    /* Get a concept from the stack here. */
    n = (rand() % SLEN) + 1;
    ScanBase = 1;
    GetOut = FALSE;
    for (ScanStack = HeadStack; GetOut == FALSE; ScanStack = ScanStack->NextInList){
        if (ScanBase == n){
            if (strncmp (ScanStack->NextInList, ScanStack->NextInList, '\0')){
                KN = ScanStack->concAddr;
                GetOut = TRUE;
            } else {
                ScanBase++;
            }
        }
    }
}

C = typealloc (Concept);
NullRL = typealloc (RelationListElt);
NullRL->PointedAt = 0;
NullRL->Domain = 0;
NullRL->TableName, '\0', 25);
NullRL->Direction = NUL;
NullRL->Used = 'N';
NullRL->NextInList = 0;
C->RelationList = NullRL;
R = typealloc (Relation);
GetOut = FALSE;
do {
    RandCard = (rand() % NumOfCards) + 1;
    FindCards = 1;
    for (ScanCards = Cards; GetOut == FALSE && ScanCards != 0; ScanCards = ScanCards->NextInList){
        if (FindCards == RandCard){
            if (strncmp (ScanCards->FromPos, KN->Label, LongestString (ScanCards->FromPos, KN->Label))
                == 0){

GetOut = TRUE;
strncpy (CLab, '0', 25);
strncpy (RLab, '0', 25);
strncpy (Referent, '0', 25);

else {
    if (strncmp (ScanCards->FromPos. KN->Label, LongestString (ScanCards->ToPos, KN->Label)) == 0)
    GetOut = TRUE;
    else {
        FindCards++;
    }
}
} while (GetOut == FALSE);

/* Add C to the graph and also to the stack, for the next iteration. */

BottomOfConcs->Next = C;
BottomOfConcs = C;
for (EndStack = HeadStack; EndStack->NextInList != 0; EndStack = EndStack->NextInList())
    NewStack = typealloc (StackElt);
    NewStack->ConcAddr = C;
    NewStack->NextInList = 0;
    EndStack->NextInList = NewStack;
    SLen++;

AllocateMapsForSimples (HeadStack->ConcAddr, Cards, FDL);

Complex = typealloc (ContextListElt);
Complex_Cxt = typealloc (Context);
Complex_SimpleGraphPtr = typealloc (ConceptListElt);
Complex->PointedAt = Complex_Cxt;
Complex->NextInList = 0;
Complex->Used = 'N';
Complex->Ident = NOTYETDEF;
Complex->Depth = 0;
Complex->Dominating = 0;
Complex->Processed = NOTYETDEF;
Complex->ListOfMembers = Complex_SimpleGraphPtr;
Complex_SimpleGraphPtr->PointedAt = HeadStack->ConcAddr;
Complex_SimpleGraphPtr->Used = 'N';
Complex_SimpleGraphPtr->Ident = NOTYETDEF;
Complex_SimpleGraphPtr->NextInList = 0;

Graph_Display_Gateway (GRAPH, '0', TLattice, RLattice);

printf (-Number of Graphs? -);
(void) scanf (-%d-,
if (NumOfGraphs < 1){
    NumOfGraphs = 1;
}

Copy = ShadowCopy (Complex, 0);
MakeData (Complex, NumOfGraphs, TDL, RDL, Complex, SOG, High);

start = times (&tmsstart);
Inquiry (Copy, TDL, RDL, Complex, SOG, TLattice, RLattice, High);
end = times (&tmsend);
printf (-\nTime taken (seconds) to perform Inquiry: \n\n-);
pr_times (end-start, &tmsstart, &tmsend);

void main ()
\begin{verbatim}
{ 
  Cardinality *DepL, **Deps;
  Defn *TDL, *RDL, *FDL;
  LatticeElt *TLatt, *RLatt;
  LatticeEltPtr *Meta;
  ContextListElt *FOG, *SOG, **First, **Second;
  int *High, NumOfDyads, *TLSize, *RLSize;
  char *Nul = "$\\0$", *StrU = "UNIVERSAL\0";

  EXEC SQL BEGIN DECLARE SECTION;
  char *Slash = "/";
  int NoOfTuples;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CONNECT :Slash;
  EXEC SQL SELECT COUNT(*) INTO :NoOfTuples FROM REPOSITORY_C;

  TDL = typealloc (Defn);
  RDL = typealloc (Defn);
  FDL = typealloc (Defn);
  strncpy (TDL->DefName, Nul, 25);
  TDL->NextInList = 0;
  strncpy (RDL->DefName, Nul, 25);
  RDL->NextInList = 0;
  strncpy (FDL->DefName, Nul, 25);
  FDL->NextInList = 0;

  FOG = typealloc (ContextListElt);
  FOG->PointedAt = 0;
  FOG->NextInList = 0;
  First = &FOG;
  SOG = typealloc (ContextListElt);
  SOG->PointedAt = 0;
  SOG->NextInList = 0;
  Second = &SOG;

  TLatt = typealloc (LatticeElt);
  RLatt = typealloc (LatticeElt);
  strncpy (TLatt->Label, Nul, 25);
  strncpy (TLatt->Label, StrU, 9);
  TLatt->SubLabels = 0;
  strncpy (RLatt->Label, Nul, 25);
  strncpy (RLatt->Label, StrU, 9);
  RLatt->SubLabels = 0;

  DepL = typealloc (Cardinality);
  strncpy (DepL->Relation, Nul, 25);
  strncpy (DepL->FromPos, Nul, 25);
  strncpy (DepL->FPRef, Nul, 25);
  strncpy (DepL->ToPos, Nul, 25);
  strncpy (DepL->TPRef, Nul, 25);
  DepL->Cardinality = MANY-MANY;
  DepL->NextInList = 0;
  Deps = &DepL;

  High = typealloc (int);
  ReadRepository (TDL, RDL, FDL, (*First), (*Second), TLatt, RLatt, (*Deps), High);

  TLSize = typealloc (int);
  (*TLSize) = 1;
  for (Meta = TLatt->SubLabels; Meta != 0; Meta = Meta->NextInList) {
    NodeCount (Meta->PointedAt, TLSize);
  }

  RLSize = typealloc (int);
  (*RLSize) = 1;
  NodeCount (RLatt, RLSize);

  printf("Maximum Number of Dyads in Test Graph? ");
  (void) scanf("%d", &NumOfDyads);
  GenSimpleGraph (TLatt, (*TLSize), RLatt, (*RLSize), NumOfDyads, (*Deps), TDL, RDL, FDL, (*Second), High);
}
\end{verbatim}