ALTERNATIVES IN VISUAL ANALYTICS AND 
COMPUTATIONAL DESIGN

by

Yingjie Victor Chen
M.Sc., Simon Fraser University, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT 
OF THE REQUIREMENTS FOR THE DEGREE OF 
Doctor of Philosophy 
in the 
School of Interactive Arts and Technology 
Faculty of Communication, Art and Technology

© Yingjie Victor Chen 2012 
SIMON FRASER UNIVERSITY 
Spring 2012

All rights reserved. 
However, in accordance with the Copyright Act of Canada, this work may be 
reproduced, without authorization, under the conditions for “Fair Dealing.” 
Therefore, limited reproduction of this work for the purposes of private study, 
research, criticism, review and news reporting is likely to be 
in accordance with the law, particularly if cited appropriately.
APPROVAL

Name: Yingjie Victor Chen
Degree: Doctor of Philosophy
Title of Thesis: Alternatives in Visual Analytics and Computational Design

Examinining Committee:

Chair: 

Steve DiPaola
Associate Professor,
School of Interactive Arts and Technology

Dr. Robert Woodbury
Senior Supervisor
Professor,
School of Interactive Arts and Technology

Dr. John Dill
Supervisor
Professor Emeritus,
School of Interactive Arts and Technology

Dr. Marek Hatala
Internal Examiner
Associate Professor,
School of Interactive Arts and Technology

Dr. Michael Terry
External Examiner
Associate Professor,
David R. Cheriton School, University of Waterloo

Date Defended/Approved: November 23, 2011
Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011
Abstract

Clearly, one’s ability to build, explore, and compare alternatives can lead to better decision making, problem solving, and design outcomes. However, I find that all too often many systems still work in a single state mode where the user can only see the result from one set of inputs at a time. Here I propose a formalism designed to represent alternatives and spaces of alternatives using the propagation-based parametric models. I choose the inputs (source nodes’ independent properties) as the representation of an alternative, which I have labeled variation heads. A variation head may contain one or several inputs to the model. The information carried by several variation heads can be unified to create a new variation head. Then I define the concept of the variation space as a collection of many variation heads. A variation space is structured by an indexed array. Two key operations, Index Unification and Cartesian Unification, can be used to unify two or more spaces. The user defines a series of variation heads as a variation space and indexes them based on his/her preference, uses unification to unify the many variation spaces to create a space of the inputs for the system, and then generates a space of results based on these inputs. This research adopts design science research methodology to iteratively refine the formalism through loops of problem awareness, design, and evaluation. A prototypical system has been developed as a formative evaluation in order to confirm, explore, and expand the formalism from a purely mathematical perspective by testing out many varied and differing kinds of data organizations. To demonstrate its usage, I show how this formalism can be used on a specific visual analytics tool (CZSaw) in order to create a space of visualization variations; I then explain both how this formalism can be used to enrich the user’s interaction in the variation space and how the indices of the space can help the user to navigate through the space.
To the sunshine of Cheryl's and my life, Sizheng Cailean Chen
“Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives.”

— William A. Foster
Acknowledgments

From the formative stages of this research to the final dissertation, I owe an immense debt of gratitude to my senior supervisor, Dr. Robert Woodbury, for his sound advice, careful guidance, persistent supervision, and kind support. I would also like to thank Dr. John Dill, the other supervisor, who lead me into the fascinating domain of visual analytics and provided me invaluable feedback and thoughtful comments throughout my Ph.D study.

I would like to extend my gratitude to Dr. Marek Hatala and Dr. Michael Terry, my thesis examiners, for a challenging and thought-provoking examination.

I offer my regards and blessings to all of my collaborators in the research and peers in the SIAT graduate program, especially the CZSaw research group, who offered their support, advice, and friendship during the past years.

Also I would like to thank NSERC and BCcampus for their financial support to let me have the freedom to conduct my Ph.D research without financial burden.

My final appreciation goes to my family. The love and support from my wife and my parents made the difficult times bearable, and my son Cailean made the good times better.
Contents

Approval ii
Abstract iii
Dedication iv
Quotation v
Acknowledgments vi
Contents vii
List of Tables xiii
List of Figures xiv

1 Introduction 1
   1.1 The Premise ................................................................. 1
   1.2 The Objective ................................................................. 2
   1.3 The Structure ................................................................. 2
       1.3.1 What Are We Trying To Do? ........................................ 2
       1.3.2 How Are We Trying To Do It? ...................................... 4
       1.3.3 How Well Are We Doing? ......................................... 5
       1.3.4 How Can We Do It Differently/Better? .......................... 6

2 The Motivation: The Parametric Model in CZSaw 8
   2.1 Propagation-based Parametric Model ................................. 9
8.2.1 Order Types .................................................. 140
8.2.2 Nodes .................................................................. 144
8.2.3 Variation Head .................................................... 145
8.2.4 Variation Space .................................................... 145
8.3 Operations on Indices ............................................... 148
  8.3.1 Index Mapping .................................................... 148
  8.3.2 Boundary of Space .............................................. 151
  8.3.3 Slice ................................................................. 154
8.4 Indexing Unification ................................................ 155
  8.4.1 Six Types of Indexing Unifications ......................... 155
  8.4.2 Indexing Unification of Several 1-d Spaces ................ 159
8.5 Cartesian Unification ................................................. 162
  8.5.1 Variation Head Cartesian With Variation Space ........ 162
  8.5.2 1-d Space Cartesian Unifications ......................... 163
  8.5.3 Several Examples of Cartesian Unifications .............. 164
8.6 Mixed Unification ................................................... 171
  8.6.1 Apply Colour Onto One Dimension ......................... 172
  8.6.2 Mix a 1-D Space With a 2-D Space ....................... 174
  8.6.3 Mix Two 2-d Spaces ............................................ 175
8.7 Order Type Unifications .......................................... 176
  8.7.1 Indexing Unification ............................................ 176
  8.7.2 Cartesian Unification ........................................... 178
  8.7.3 Cartesian Square ............................................... 182
8.8 Relational Algebra ................................................... 183
  8.8.1 Select .............................................................. 183
  8.8.2 Rename ............................................................ 184
  8.8.3 Projection .......................................................... 185
8.9 Summary .............................................................. 186

9 CZSaw Use Cases ..................................................... 187
  9.1 Use Case A: VAST 2008 Mini Challenge 3 - Cell Phone Calls .... 188
    9.1.1 Symbolic Model of the Solution ......................... 188
    9.1.2 Possible Interactions ....................................... 193
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Main types of research questions. Adapted from (Meltzoff, 1998)</td>
<td>82</td>
</tr>
<tr>
<td>9.1</td>
<td>Screenshots generated by CZSaw from the cellphone variation space</td>
<td>193</td>
</tr>
<tr>
<td>9.2</td>
<td>The result of Cartesian unification. Tuples are {originalPhone, objectType, relationThreshold}</td>
<td>194</td>
</tr>
<tr>
<td>9.3</td>
<td>Select variation heads that subsumed by {originalPhone=1}</td>
<td>196</td>
</tr>
<tr>
<td>9.4</td>
<td>Select variation heads that subsumed by {objectType: tower, relationThreshold:-1} or {objectType: phone, relationThreshold:-2}</td>
<td>198</td>
</tr>
<tr>
<td>9.5</td>
<td>Closely related variation heads of {originalPhone:2, objectType: phone, relationThreshold:-2}</td>
<td>199</td>
</tr>
<tr>
<td>9.6</td>
<td>Hysterical space created from selected heads</td>
<td>201</td>
</tr>
<tr>
<td>9.7</td>
<td>All possible social networks</td>
<td>212</td>
</tr>
<tr>
<td>9.8</td>
<td>Closely related results by expanding the parameter range</td>
<td>213</td>
</tr>
<tr>
<td>9.9</td>
<td>A sliced space from picked variation heads</td>
<td>214</td>
</tr>
<tr>
<td>9.10</td>
<td>Studying the effects of changing three parameters</td>
<td>215</td>
</tr>
<tr>
<td>9.11</td>
<td>Unifying two chosen results</td>
<td>217</td>
</tr>
<tr>
<td>9.12</td>
<td>A Cartesian square of selected results</td>
<td>218</td>
</tr>
<tr>
<td>9.13</td>
<td>Entities related to a single day of April</td>
<td>222</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 A DAG of a parametric model ........................................ 11
2.2 The general structure and dataflow of CZSaw .......................... 17
2.3 The CZSaw script to search for replacement phone .................. 21
2.4 Left: search for phone 5; Right: phone 5 is displayed in the graph view .. 22
2.5 Related phones of phone 5 .................................................. 23
2.6 Phones that connect to at least three of the related phones .......... 23
2.7 The symbolic model created from the script of section 2.2.2 ........... 24
2.8 Phone 309 is likely the replacement of phone 1 ....................... 25

3.1 VisTrails: different visualizations are generated by different versions of dataflows .................................................. 33
3.2 Tableau: Compare product sales and gross profit (Heer et al., 2008) .... 35
3.3 Jigsaw: Two graph views in one application to explore the data ....... 38
3.4 GeoTime: Left: zoom in to a smaller time range. Right: other events links to the main plot (Oculus, 2010) .................. 40
3.5 Photoshop: “Variations” operation ...................................... 42
3.6 Photoshop: Use of combinations of layers to retrieve alternatives. Courtesy York & Chapel, Inc. .............................................. 43
3.7 Photoshop: left palette - the history palette. Right photo is partially rolled back by the history brush .............................. 45

4.1 Git: object hierarchy and linkages .................................... 53
4.2 Git Workflow. Each node represents a revision. Edges point from the parents to the children. ................................. 54
4.3 Merge vs. Rebase. C₅ has the same content as C₄′ .................. 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Typed Feature Structure: A single-fronted cottage house, its plan, and partially represented feature structure (Woodbury et al., 1999, 2000)</td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>AND/XOR tree: a generic structure for characterizing variety (Jiao et al., 2000, 2007)</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>A configuration space (Jiao et al., 2007)</td>
<td>65</td>
</tr>
<tr>
<td>4.7</td>
<td>The genetic algorithm. Adapted from (Gen and Cheng, 2000)</td>
<td>66</td>
</tr>
<tr>
<td>4.8</td>
<td>Parallel Paths: Process diagram shows clear branching in Parallel Pie tasks</td>
<td>69</td>
</tr>
<tr>
<td>4.9</td>
<td>A subjunctive interface of querying census data (Lunzer and Hornbæk, 2008)</td>
<td>74</td>
</tr>
<tr>
<td>4.10</td>
<td>The data flow from combining parameters to generate results.</td>
<td>76</td>
</tr>
<tr>
<td>5.1</td>
<td>Reasoning on design cycle. Adapted from (Takeda et al., 1990)</td>
<td>81</td>
</tr>
<tr>
<td>6.1</td>
<td>Variation properties and variation head of a graph</td>
<td>90</td>
</tr>
<tr>
<td>6.2</td>
<td>Variation compatible graphs</td>
<td>91</td>
</tr>
<tr>
<td>6.3</td>
<td>The AVM notation of a variation head with two nodes</td>
<td>92</td>
</tr>
<tr>
<td>6.4</td>
<td>Assigning variation heads into the space V</td>
<td>98</td>
</tr>
<tr>
<td>6.5</td>
<td>1-D (a), 2-D (b), and 3-D (c) variation space</td>
<td>99</td>
</tr>
<tr>
<td>6.6</td>
<td>Sparse variation spaces - 1D and 2D, and their index sets</td>
<td>100</td>
</tr>
<tr>
<td>6.7</td>
<td>Index Mapping to move elements in two dimensions. $I_i$ and $I_j$ represent indexes of two dimensions.</td>
<td>102</td>
</tr>
<tr>
<td>6.8</td>
<td>Unify two 1-D spaces by matching indices</td>
<td>103</td>
</tr>
<tr>
<td>6.9</td>
<td>Six ways of indexing unification</td>
<td>105</td>
</tr>
<tr>
<td>6.10</td>
<td>Index shifting and index unification to append two 1-D spaces</td>
<td>106</td>
</tr>
<tr>
<td>6.11</td>
<td>Indexing unification with different order types</td>
<td>107</td>
</tr>
<tr>
<td>6.12</td>
<td>Cartesian unification on two 1-D spaces</td>
<td>108</td>
</tr>
<tr>
<td>6.13</td>
<td>Cartesian Unification on two 2-D spaces</td>
<td>109</td>
</tr>
<tr>
<td>6.14</td>
<td>The result of Cartesian unification may contain many $\top$</td>
<td>110</td>
</tr>
<tr>
<td>6.15</td>
<td>Cartesian square unification of a 1-D space - indices of heads that need to be unified</td>
<td>113</td>
</tr>
<tr>
<td>6.16</td>
<td>A 1-D space with multiple-property heads</td>
<td>114</td>
</tr>
<tr>
<td>6.17</td>
<td>Cartesian unification of multiple-property heads. Values unify to $\top$ if two values are not equal.</td>
<td>115</td>
</tr>
<tr>
<td>6.18</td>
<td>Decomposition of a 2-D space</td>
<td>116</td>
</tr>
<tr>
<td>6.19</td>
<td>Projection of the resulting space of Figure 6.17 to single properties</td>
<td>117</td>
</tr>
</tbody>
</table>
6.20 Flattening one result space of figure 6.19 .................................................. 118
6.21 Unify along the 2nd dimension (column) of a space ................................. 119
6.22 Unify 1-D space carrying values of ascending numbers .......................... 119
6.23 How to combine these two spaces? .............................................................. 120
6.24 Append B at the end of the first dimension of A ....................................... 121
6.25 Append B at the end of the second dimension of A .................................... 121
6.26 Append B at the end of both dimensions of A ............................................ 122
6.27 The hysterical space from the input of Figure 6.23 ................................... 123

7.1 An relation example - 4 domains and the table view. Null means that piece
of information is unknown/missing ................................................................. 126
7.2 A relation directly mapped from a variation space .................................... 127
7.3 A relation view of a variation space .......................................................... 127
7.4 Truth table of Null and ⊥ ......................................................................... 128
7.5 Cartesian unification and θ unification ......................................................... 133
7.6 θ Unification and outer unifications ............................................................ 135

8.1 Screenshot of prototype I .............................................................................. 138
8.2 Order relation of lifted integers .................................................................. 141
8.3 Order of ascending integers ....................................................................... 142
8.4 Order of descending integers ....................................................................... 143
8.5 A variation head and its 3 nodes in 3D ......................................................... 147
8.6 A variation space in 3D .............................................................................. 147
8.7 The input of index mapping examples ......................................................... 148
8.8 Index shifting examples .............................................................................. 149
8.9 Index distribution examples ........................................................................ 150
8.10 Append and prepend one dimension ............................................................ 151
8.11 Boundaries of a 2-D space .......................................................................... 152
8.12 A 3D space for selecting boundary ............................................................. 153
8.13 Retrieve a subspace by slicing at given index ............................................. 154
8.14 Inputs of indexing unification ..................................................................... 155
8.15 Union indexing unification ........................................................................ 156
8.16 Intersection indexing unification ................................................................. 156
8.17 Left indexing unification ............................................................................ 157

xvi
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.18 Right indexing unification</td>
<td>157</td>
</tr>
<tr>
<td>8.19 Left difference unification</td>
<td>158</td>
</tr>
<tr>
<td>8.20 Right difference unification</td>
<td>158</td>
</tr>
<tr>
<td>8.21 Cartesian unification result</td>
<td>159</td>
</tr>
<tr>
<td>8.22 Inputs of indexing unification example</td>
<td>159</td>
</tr>
<tr>
<td>8.23 Indexing unification example</td>
<td>160</td>
</tr>
<tr>
<td>8.24 Different ways of combining two spaces</td>
<td>161</td>
</tr>
<tr>
<td>8.25 Cartesian unification of a variation head to a space</td>
<td>162</td>
</tr>
<tr>
<td>8.26 Cartesian unification of 1-D spaces</td>
<td>163</td>
</tr>
<tr>
<td>8.27 Cartesian unification example I - inputs</td>
<td>164</td>
</tr>
<tr>
<td>8.28 Cartesian unification example I - results</td>
<td>165</td>
</tr>
<tr>
<td>8.29 Cartesian unification example II</td>
<td>166</td>
</tr>
<tr>
<td>8.30 Cartesian unification example III</td>
<td>167</td>
</tr>
<tr>
<td>8.31 Cartesian unification example IV</td>
<td>168</td>
</tr>
<tr>
<td>8.32 Cartesian unification example V</td>
<td>169</td>
</tr>
<tr>
<td>8.33 Cartesian unifications vs. indexing unifications</td>
<td>170</td>
</tr>
<tr>
<td>8.34 Mixing example - applying colour on one dimension</td>
<td>173</td>
</tr>
<tr>
<td>8.35 Mix a 1-D space with a 2-D space</td>
<td>174</td>
</tr>
<tr>
<td>8.36 Mix two 2-D spaces</td>
<td>175</td>
</tr>
<tr>
<td>8.37 Indexing unification of heads with ascending or descending integers</td>
<td>177</td>
</tr>
<tr>
<td>8.38 Unify two variation heads to a 2-D space with order types</td>
<td>178</td>
</tr>
<tr>
<td>8.39 Inputs of 1-D order type Cartesian unification</td>
<td>179</td>
</tr>
<tr>
<td>8.40 Cartesian unification of two 1-D spaces with order types</td>
<td>180</td>
</tr>
<tr>
<td>8.41 Another example of order type Cartesian unification</td>
<td>181</td>
</tr>
<tr>
<td>8.42 Order type Cartesian square unification</td>
<td>182</td>
</tr>
<tr>
<td>8.43 Select variation heads from a space</td>
<td>183</td>
</tr>
<tr>
<td>8.44 Rename name of nodes and properties</td>
<td>184</td>
</tr>
<tr>
<td>8.45 Project a variation space</td>
<td>185</td>
</tr>
<tr>
<td>9.1 The parametric model to find the replacement phone</td>
<td>189</td>
</tr>
<tr>
<td>9.2 Two states of using CZSaw to find replacement phones</td>
<td>190</td>
</tr>
<tr>
<td>9.3 Example data of the VAST 2009 mini challenge 2</td>
<td>202</td>
</tr>
<tr>
<td>9.4 The parametric model for the 2009 social network challenge</td>
<td>205</td>
</tr>
</tbody>
</table>
9.5 Using CZSaw to solve the VAST 2009 social network challenge  
9.6 The parametric model and one state of finding entities related to the Dubai conference  
9.7 Locations related to all April days  
9.8 People related to all April days  
9.9 Reports related to all April days  
9.10 Organizations related to April days  
10.1 MobileAnalymator: Blogs highlighted by different keywords  
10.2 Parametric models with disabled nodes. Are these models at left equivalent to models at right?  
10.3 Change the value of a graph dependent property  

A.1 A variation head and its 3 nodes in 3D (Figure 8.5)  
A.2 A variation space in 3D (Figure 8.6)  
A.3 The input of index mapping examples (Figure 8.7)  
A.4 Index shifting examples (Figure 8.8)  
A.5 Index distribution examples (Figure 8.9)  
A.6 Append and prepend one dimension (Figure 8.10)  
A.7 Boundaries of a 2-D space (Figure 8.11)  
A.8 A 3D space for selecting boundary (Figure 8.12)  
A.9 Retrieve a subspace by slicing at given index (Figure 8.13)  
A.10 Inputs of indexing unification (Figure 8.14)  
A.11 Union indexing unification (Figure 8.15)  
A.12 Intersection indexing unification (Figure 8.16)  
A.13 Left indexing unification (Figure 8.17)  
A.14 Right indexing unification (Figure 8.18)  
A.15 Left difference unification (Figure 8.19)  
A.16 Right difference unification Figure 8.20)  
A.17 Cartesian unification result (Figure 8.21)  
A.18 Inputs of indexing unification example (Figure 8.22)  
A.19 Indexing unification example (Figure 8.23)  
A.20 Different ways of combining two spaces (Figure 8.24)  
A.21 Cartesian unification of a variation head to a space (Figure 8.25)
A.22 Cartesian unification of 1-D spaces (Figure 8.26) ................. 248
A.23 Cartesian unification example I - inputs (Figure 8.27) ............... 249
A.24 Cartesian unification example I - results (Figure 8.28) ............... 250
A.25 Cartesian unification example II (Figure 8.29) ....................... 250
A.26 Cartesian unification example III (Figure 8.30) ....................... 251
A.27 Cartesian unification example IV (Figure 8.31) ....................... 252
A.28 Cartesian unification example V (Figure 8.32) ....................... 253
A.29 Cartesian unifications vs. indexing unifications (Figure 8.33) ......... 254
A.30 Mix a 1-D space with a 2-D space (Figure 8.35) ..................... 255
A.31 Mix two 2-D spaces (Figure 8.36) .................................... 256
A.32 Indexing unification of heads with ascending or descending integers (Figure 8.37) ................................................................. 257
A.33 Unify two variation heads to a 2-D space with order types (Figure 8.38) .... 258
A.34 Inputs of 1-D order type Cartesian unification (Figure 8.39) ......... 259
A.35 Cartesian unification of two 1-D spaces with order types (8.40) ......... 260
A.36 Another example of order type Cartesian unification (Figure 8.41) .... 261
A.37 Order type Cartesian square unification (Figure 8.42) ............... 262
A.38 Select variation heads from a space (Figure 8.43) ..................... 263
A.39 Rename name of nodes and properties (Figure 8.44) .................. 264
A.40 Project a variation space (Figure 8.45) .................................. 265
Chapter 1

Introduction

1.1 The Premise

Many computing systems provide interfaces for the user to input parameters in order to control the behavior and appearance of the system and its results. But most of them only support a single state; in other words, these can only take one set of inputs at a time and only produce one result. The user has to iteratively input other values to see alternatives. Sometimes there are so many alternatives that, eventually, the user may forget choices he/she has considered, and the comparison of these states becomes even more difficult. A simple example is the class of flight booking systems. By entering the departure and destination cities and dates, these systems are able to list a series of tickets from which the user may choose. However, with such systems, the user can only input one set of parameters a time. Although some flight booking systems can provide the user with a broader choice by integrating flights from multiple airlines, or by expanding the flight date to cover a wider range, the user’s control is still limited. For a truly flexible travel plan, the user may have multiple choices of destination cities and flexible dates available. The final decision depends on comparison - e.g., the balance between the rate, season, and location. These parameters are highly intentional. They totally depend upon the user’s choices, and the ordering of alternatives may be important due to the user’s preferences. Even if there are only three parameters to which the user may apply values (destination, departure date, and returning data), the number of combination possibilities can be large. Each combination of the parameters results in a different solution. At this point, we shall define these different solutions as alternatives. People solve problems by searching in the
problem space (Newell and Simon, 1972). These alternatives simply construct a problem space. I believe that an effective way of managing these alternatives can lead to better decision making, problem solving, and design outcomes.

1.2 The Objective

George (1980) listed five obvious problems that arise when analysts make decisions; all are related to their inability to access alternatives in a broad manner (e.g., Select first answer that appears “good enough”). Thomas and Cook (2005) categorized several analytical tasks that visual analytic tools should support: understanding past and present situations, identifying possible alternative futures, monitoring current events for warning, determining indicators of intent, and supporting decision making. All these tasks imply the need to handle alternatives.

This PhD dissertation aims to establish a formalism to represent alternatives, enable developers to embed such a formalism into their system designs, and provide users with a means to set up, navigate, discover, and compare alternatives.

1.3 The Structure

Catherine Marshall outlines four key questions underlying the research process (Marshall, 2006):

1. What are we trying to do?
2. How are we trying to do it?
3. How well are we doing?
4. How can we do it differently/better?

The answers to these questions establish the structure and content of my dissertation work.

1.3.1 What Are We Trying To Do?

We can see alternatives from two perspectives. First, for a given problem, there may exist multiple alternative results. While even the procedure and underlying logic of the solution
may be the same, according to the variety of input parameters, the result may be different. Due to the number of inputs, many problems present a vast range of possibilities. Among these alternatives, there may be an optimal result. AI (Artificial Intelligence) algorithms may assist in the discovery of an optimal result if there exists a single objective standard. But many problems do not provide such a standard for comparison. They are largely reliant on human judgement. We need an efficient way to represent alternatives to support subjective comparison.

From another perspective, an alternative may be seen as partial information that only represents one part of the whole problem. To perceive the whole picture the user may have to unify multiple alternatives. In many real world critical tasks, we cannot afford to miss any chance (e.g., analysis of terrorist attacks); we have to cover as many options as possible. A human being’s cognitive ability is limited. Simon (1955) argued for bounded rationality, noting that human beings encounter cognitive and temporal limits in the process of receiving, storing, and recalling information. It thus becomes important to have an efficient mechanism to allow people to study a large numbers of alternatives as well as the many varied inputs involved in generating these alternatives.

Researchers have worked to integrate different sets of parameters and results on the interface level. Design Galleries (Marks et al., 1997) automatically generate and lay out parametric variations (e.g., 3D renderings with different light settings). Parallel Paths & Pies (Terry et al., 2004) use a set-based interface for the user to explore alternatives while manipulating images. Subjunctive interfaces (Lunzer and Hornbæk, 2008) allow parallel exploration of multiple scenarios during information exploration. Spreadsheet interface (Jankun-Kelly and Ma, 2000) treats visualization exploration as a process of examining a multi-variable space of parameter values. More recently, the Juxtapose editor (Hartmann et al., 2008) provides an environment for designing multiple alternatives for both application logic and interface parameters.

Also, researchers have developed formalisms to represent alternatives in many varied areas. Typed feature structure, a general purpose data structure designed to identify and group individual features, is used to represent design variations in design space exploration (Woodbury et al., 1999). The AND/XOR Tree is used to represent the structure of a product and its components in product family design (Jiao and Tseng, 1999a). Directed acyclic graph (DAG) is utilized in software configuration management (Torvalds and Hamano, 2010a) to represent branches and versions of software development.
This research establishes a formalism that can represent alternatives and support users’ interactions with alternatives through embedding the formalism into a system design. It will be very hard (if not impossible) to formalize alternatives in general. Currently, I am interested in exploring different scenarios that differ only in their inputs. These inputs could either be set up by the user intentionally or generated from the given conditions of the problem.

1.3.2 How Are We Trying To Do It?

Alternatives can be further distinguished into variations and alternatives. For a given problem or design, we call different solutions (e.g., differing on the procedure, or the algorithm of solving the problem) alternatives. For the same procedure or solution, it may be applied with different parameters. We define these similar solutions with differing parameters as variations.

Focusing further, this formalism is built based on a propagation-based system (Aish and Woodbury, 2005) that uses a graph-based approach to represent objects as nodes and constraints as links in a graph. The user organizes this graph to directly solve a problem. This graph divides a problem into easily solvable subproblems, and then solves these problems and composes their answers into a complete solution (Hoffman and Joan-Arinyo, 2005; Woodbury, 2010). The representation of a propagation-based system is a directed acyclic graph that has some nodes (or properties inside these nodes) as inputs. In the parametric modeling system GenerativeComponents™, the graph is called symbolic model. In the visual analytics system CZSaw we developed, the graph is called dependency graph. The propagation mechanism ensures the consistency of the whole model: whenever an inputting node changes its value, all of its downstream nodes are evaluated and populated with proper values. In our research, we should call a collection of input nodes and properties a variation head. This dissertation aims to provide a general solution to organize these variation heads and to manipulate collections of variation heads.

Through unification, information carried by two objects can be unified to create a new object (Carpenter, 1992). Similarly, we can combine two sets of input nodes (variation heads) to create a new set of nodes. A collection of ordered variation heads is called a variation space, and such a variation space has a structure that is similar to a multi-dimensional indexed array. Our premise is that by defining a structure of the space and
algorithms to design and edit such structure, we are able to give system developers a means to setup, explore, and generate alternatives that will benefit the end users. Furthermore, we are able to see the connections between our variation space and relational algebra. Operations from relational algebra and relational database management systems (RDBMS) can further enrich the formalism.

1.3.3 How Well Are We Doing?

To explore and demonstrate the characteristics of this formalism, a functional prototype system has been developed to simulate and test. This prototype uses 3D geometry data (X, Y, and Z, plus color and shape to enable handling five properties) to directly visualize variation spaces as grids of nodes in 3D. This allows us to visually inspect and understand both the structure of variation spaces and the results of manipulation.

This formalism has to be applied to real (or at least close to real) data in order to prove its usefulness. VAST Challenges (VAST, 2008, 2009, 2010) provide us with professionally designed simulation data. Our SFU CZSaw research group has worked to develop CZSaw, a visual analytic system, to help analysts make sense of text documents, entities, and relations. CZSaw has demonstrated its capability with handling these challenges, especially when analyzing entities and relations.

CZSaw can be seen as a propagation-based parametric design system for visual analytics. CZSaw enables the user to create a problem solver by capturing a series of user interactions and building a symbolic model. The CZSaw script allows the user to refine and reuse the recorded analysis process. Changing parameter values used in interactions will allow an analyst to explore alternatives without repeating the whole analysis process. This propagation-based model makes alternative exploration much easier. In CZSaw, the data, visualization of data, and windows to display visualizations are all objects in the model. We can simply change values of the model’s inputs; the propagation mechanism will then ensure that all related nodes in the model get updated, which will result in a different visualization for the user to analyze.

Our formalism allows the user to manage a large number of alternatives in CZSaw. I describe how this formalism can be used in CZSaw to create a space of visualization variations by demonstrating the formalism with three VAST challenges. The user simply assigns a series of values for each of the inputting parameters in the model. The formalism
creates the variation space through indexing unification and Cartesian unification. The application of one variation head on CZSaw’s symbolic model will drive CZSaw to create one set of visualizations. The application of a space of variation heads onto the symbolic model will generate a space of visualization results. With an appropriate user interface (not included in this study), the analyst then sees many visualization variations at once. He/she can examine these visualization variations to discover the best solution.

1.3.4 How Can We Do It Differently/Better?

This formalism also enriches the user’s interaction within the space. First, the analyst inspects and compares different visualizations in the space, which may give the analyst a better understanding of the problem. Second, the analyst may interact with the visualization space by selecting one or several results. Here I propose some examples of possible new spaces generated from such interactions:

- The new space may contain several results that are closely related to the selected results (from the perspective of inputting parameters).

- The space could contain results that only differ in one parameter so that the analyst can study the effect of said one parameter change.

- The space may contain results that are blended with the selected results (the new results’ input parameters are generated from inputs of all selected results).

- By using an order type unification (Chang, 1999), we can generate results that contain more restricted information than the selected results.

- We may be able to combine many simple visualizations, such as those which permit us to integrate many simple node-link graphs into a larger network.

The implementations and evaluations of these interactions highly rely on the context of tasks and the applications. This part will be conducted in my future research. Although I have proposed many possible ways to interact with variation spaces and individual variation heads and results in those spaces, I believe there remains much more for us to explore. For example, the order type unification can provide us with an almost endless number of variation heads from unification. But I can only scratch its surface with some introductory descriptions and simple examples. As for the symbolic model, currently I
shall only work with the variation heads of a given symbolic model. The symbolic model does not change. However, in the real world, one problem may have many different solutions, which may be represented by different symbolic models.

Informed by the research findings, the fourth question requires a reevaluation of research purpose and goals. Instead of introducing new plans and actions, my final chapter summarizes current research findings, discusses potential extensions, and proposes future directions.
Chapter 2

The Motivation: The Parametric Model in CZSaw

This research was initiated and motivated by our (the Simon Fraser University Visual Analytics research group) development of CZSaw (Kadivar et al., 2009). CZSaw is a visual analytics system designed to help analysts analyze entities, relations, and text documents. Specifically, the achievements delivered and problems solved through the CZSaw’s parametric model have directly inspired my research. In CZSaw, the parametric model is called dependency graph. This chapter introduces the theoretical context of the dependency graph, our dependency graph design in CZSaw, and its performance with an analysis example. In 2010, CZSaw participated in the VAST contest (VAST, 2010) and won VAST 2010 Mini Challenge 1 Award: Outstanding Interaction Model (Chen et al., 2010b).

CZSaw focuses on supporting the analysis process. We designed a script language to describe and capture the user’s interactions and created a parametric model (Woodbury, 2010) to symbolize the logic of analysis process. The parametric model is called dependency graph in CZSaw. The scripts capture all of the meaningful user interactions. For example, while creating a new graph view window, all parameters of the view were captured. Thus, a replay of the script will let us review the whole analysis process, including creating views, exploring the graph in views, etc. The dependency graph is a representation of the propagation-based parametric design system, in the form of a directed acyclic dependency graph that is capable of propagating changes (Woodbury, 2010). Nodes in
the graph (variables in CZSaw) are results generated through user interactions. The analyst works on the data by interacting with entity or relation variables in views. Each step creates new results, which are represented as new nodes in the graph. Edges indicate dependency relationships among variables. Any content change to a variable triggers the propagation mechanism to update downstream variables and, in turn, update data views to reflect the change. The graph’s root node represents the entire data set. Thus, any change to the data set (such as an entity refinement operation) starts the propagation at the root node, potentially changing all analysis results and updating all data views. The analyst can also reuse parts of the analysis process by assigning a new value to one node in the graph.

In essence, CZSaw is an analytical system which allows analysts to visually design a solution to solve a problem. As a parametric system, CZSaw easily encourages analysts to change, reuse, explore, and search for new solutions. This dependency graph provides a logic map for analysts to navigate and interact.

2.1 Propagation-based Parametric Model

2.1.1 Parametric Design

Parametric design was introduced into the commercial computer-aided design (CAD) systems by Pro/Engineer (Parametric Technology Corporation) in the 1980s. In the domain of mechanical engineering, parametric design is now the standard for CAD. Since the 2000s, it started to influence other business domains, such as architecture (e.g., GenerativeComponents for MicroStation from Bentley Systems Inc., AutoCAD from AutoDesk Inc.) and industrial design (e.g., SolidWorks of Dassault Systèmes SolidWorks Corporation). Parametric design systems model a design as a constrained collection of schemata (objects containing variables and constraints amongst the variables). Designers work in such systems at two essential levels: (1) defining schemata and constraints, and (2) searching within a schema collection for meaningful instances (Aish and Woodbury, 2005).

Hoffman and Joan-Arinyo (2005) worked to define parametric systems through a constraint solving approach. The graph-based approach uses a graph to represent the constraint problem. Nodes are objects, and edges are constraints among the objects. The solver analyzes the graph and formulates a solution. A logic-based approach translates
the problem into a set of assertions and axioms characterizing the constraints and the objects. Algebraic methods directly translate the constraint problem into a set of nonlinear equations. Solving a chosen problem becomes a process of choosing the proper methods to solve these nonlinear equations.

Hoffman and Joan-Arinyo (2005) further divided the graph-based approach into three categories: constructive, degree of freedom analysis, and propagation. The constructive approach solves the problem as a symbolic sequence of basic construction steps. Each step works on a subset of objects by taking a rule from a set of predefined operations. The degree of freedom analysis labels the edges of the graph with the numbers of degrees of freedom. The degree is canceled by the associated constraints of vertices (objects) in the graph. According to the propagation method, objects and constraint become the variables and equations of algebraic equations. Vertices are variables. Equations connecting input and output variables become the edges that point from the input variables to the output variables. Propagation methods proceed from upstream known information to downstream unknowns, solve these equations and evaluate variables based on the orientations of the edges. However, the propagation methods may fail to find a solution if there is a loop along the edges (starting from one node $V$, and following a sequence of edges that eventually loops back to $V$).

Parameterization increases the complexity of design. In order to create a desired artifact, the designer has to initially design a conceptual structure. For a designer, parameterization can have both positive and negative effects. According to Aish and Woodbury (2005), parameterization can enhance search for designs better adapted to context, can facilitate discovery of new forms and kinds of form-making, can reduce the time and effort required for change and reuse, and can yield better understandings of the conceptual structure of the artifact being designed. However, apart from these positive effects, parameterization may require additional effort through increasing the complexity of local design decisions and increasing the number of items to which attention must be paid in task completion.

Propagation based systems are the most simple type of parametric systems (Aish and Woodbury, 2005; Woodbury, 2010). It is easy for the end user to build the conceptual structure of the design, and its algorithms are also efficient for the system to solve. In most systems, the directed graph is organized by the user to directly solve a problem (or design). A well-formed design should avoid loops, which will guarantee that the solution
is found. The representation of the propagation based system is called parametric model.

2.1.2 The Parametric Model

A parametric model can be represented by a directed acyclic graph (DAG) (Figure 2.1.) (Woodbury, 2010). A DAG is a graph containing nodes and edges with directions. The edge links from a predecessor node (upstream) and points toward a successor node (downstream). A sequence of nodes that link from one to another in a single direction forms a path. The terms “downstream” and “upstream” refer to nodes that occur in at least one path of the given node. A well-formed model has no cyclic paths. A node is a schema, an object containing properties. Each property has an associated value. A value could be an atomic value (e.g., integer, string) or a complex object. A node having only one property is a single-property node. In the following sections, most often the term nodes’ values are reference to single property nodes with the single property value.

![Diagram](image)

Figure 2.1: A DAG of a parametric model

Each node in this graph has one (and only one) unique name (or ID). The user and the system access the node by referring to its name.

A node’s (the successor) property value can be derived from other nodes’ (predecessors) properties. Such a value assigned by a constraint expression and computed by evaluating the constraint expression. It is called graph-dependent because the evaluated result depends on the values of its predesign nodes. A constraint expression is a well-formed formula comprising objects, function calls, and operators. The constraint expression defines edges among nodes. The direction is derived from the predecessor to the successor node, indicating that the successor node (property) holds a constraint expression that
uses the predecessor node (property). We also conventionally say that data flows from predecessor nodes to successor nodes when the constraint expression of successor nodes evaluated. In a number of cases a successor may have multiple predecessors, which will result in multiple edges all pointing toward the successor. The system ensures that the whole model is consistent. Whenever the system changes a node’s value, the system will evaluate all of the node’s downstream nodes (i.e., evaluating constraint expressions and populate results).

A property may hold an explicit value (where it does not depend on other properties). Such a property is then known as graph-independent. Otherwise, the property is called graph-dependent, which means that its value depends on other properties.

Depending on the values of its properties, a node can be a source node, sink node, or internal node. A source node is a node with no graph dependent property. Data only flows out of such node to its successor nodes. A sink node is a node with no successor; its properties have never been used in other nodes’ (properties’) constraint expressions. Data flows into the sink node but no data flows out. An internal node is a node in the middle of the graph, and it is both a predecessor and successor. In Figure 2.1, A, B, and C are source nodes; U and V are internal nodes; W is a sink node.

To record a node’s value, we use the format of the node name and property value pairs as nodeName:{property=value}; for example, A:{p=3, q=4}. An alternative format could be in dot notation, such as {nodeName.property= value}, e.g., {a.p=3, a.q=4}.

In the parametric modeling system GenerativeComponents™, the visualization of the parametric model is called Symbolic Model. In CZSaw, the visualization is called dependency graph.

2.1.3 State of a Parametric Model

A state is a parametric design or one solution to the problem. A state of a parametric model is the tuple of values (explicit value or constraint expression) of all nodes. A state for a given node in the parametric model comprises its values and the values of all upstream nodes. Given explicit values of graph-independent properties, all graph-dependent properties are evaluated automatically by the system.

The value (explicit value or constraint expression) can be seen as the information held by the property. The lowest information value is no information. In some cases, the
information could be inconsistent.

A property may have no information (where we do not know its value). In such cases, we call it unbound and denote as $\bot$. If a graph dependent property uses $\bot$ in its constraint expression, depending on the operator, the evaluated result may or may not be $\bot$. A state with unbound values could be seen as missing information. Such a state represents the partial information of a full state.

Another value of a property represents inconsistent information. We call such a value top and denote it as $\top$ (Ait-Kaci, 1984). If a graph dependent property uses $\top$ in its constraint expression, depending on the operator, the evaluated result may or may not be $\top$.

Assigning different values to some nodes in the parametric model will create a new state. If we only change values of graph independent properties, the topological structure of the graph will be preserved, but the evaluated values of successor nodes will be updated. From this perspective, the user has created a mathematical model in a design and is able to create new design outcomes by reusing this model with different parameter values. The user may also re-define constraint expressions for successor nodes. By doing so, the user creates a different model, but with a possibly similar set of source nodes.

### 2.2 The Parametric Model in CZSaw

#### 2.2.1 CZSaw Scripts

For a given analytic problem, different people may each have a different approach to solve the problem. Such approaches normally follow logical procedures that embed some problem solving strategies. This logical procedure is an analysis process that can be captured as a series of user interactions.

CZSaw provides the interaction model and history necessary to support the analysis process. One component of this is a method for analysts to record, and then reuse or replay their analysis steps in order to help them better understand, explore, reference, and reuse their analysis. CZSaw has a script language with commands to perform simple analytics tasks. User interactions are recorded and translated into lines in a script.

We want the analyst to be able to interpret the analysis procedure by reading the text-based scripts in a manner just like reading a story. The story is the analysis process.
Its components are the steps taken in CZSaw. Reading the story may assist the analysts to better understand how and why they have arrived at a result and how accurate or significant it might be.

Based on this idea, we refined the scripting language for performing operations meaningful to the analysis process. The language consists of commands that an analyst may enter directly. The analyst could manually edit the script by using any text editor. This process provides analysts with additional programming power to solve complex problems. Of course, analysts could also interact directly with data views. Each such user interaction is translated into a block of one or more commands that is termed a transaction. Thus, analysts perform the analysis process by either interacting with views or by directly typing script commands into the dialog box.

Blocks of scripts are grouped as transactions. Transactions occur at the level of meaningful task elements completed within the system. Currently, only actions that modify the visual or data model of the system are recorded. Micro interactions, such as moving the mouse and keying in characters, are not recorded. Thus the script is not a like a movie that fully records everything happening in the process, but rather that records at task level.

When the analyst interacts with the system to execute one operation (e.g., search for entities and display the result in a view), a block of commands is created and recorded as one transaction. Some script commands such as search and relatedNodes describe system actions on the data structures. Other commands describe system actions that directly result in data view changes such as showNodes and hide. Commands in this second group will be interpreted differently depending on the particular data view.

The script is not just a passive recording of system actions; instead, the script is actually what drives CZSaw. Therefore, rerunning this script step by step will replay the whole analysis process. Data views are updated exclusively by the script, regardless of whether the commands were the result of the analyst’s interactions with the specific data views or whether the commands had merely been typed directly into the script. We can treat this script as a program created by the analyst while using CZSaw. Editing it allows for quick refinement upon the analysis process, since any later steps after the edit do not have to be retaken by the analyst. In this way, the script can be modified to check alternatives or fix mistakes. An important aspect of the script is that it permits analysis interactions to be reusable, especially since analysis processes often consist of many repetitive actions
(Shrinivasan and van Wijk, 2008). Additionally, an analyst may find exploration patterns applicable to different data sets or an updated version of the same data set. Such tasks are easily accomplished by simply extracting the appropriate set of script commands and running them on different data sets. Suitably tested script blocks should decrease errors in sequences of analytical actions.

Within a script, new data can be added, such as adding a new report just received by the analyst. In the real world, data is dynamic, growing, and changing. Current analysis results may change, or particular road blocks might be removed. Adding new data and rerunning the script allow for adaptation to such data changes which can reveal new results. Also, it is possible to explore further alternatives by adjusting the parameters in the script. For example, an analyst could iterate a predefined analytical process over a group of objects representing people, places, or things.

In CZSaw, analysts who wish to directly program part of the script may do so in the script view window. The script view is where the analyst can create, copy, edit, and run script commands. The analyst can also use any favorite text editor to “program” the script. Thus, CZSaw enables a much greater control over the analysis process itself than would be possible using only the data views. To achieve alternative results, the analyst can simply change parameter values used in interactions and replay the script.

In the current version, CZSaw embeds BeanShell (BeanShell, 2010) as its scripting engine. Other than executing standard Java syntax, BeanShell also supports common scripting conveniences such as loose types, commands, and method closures. This BeanShell engine allows the CZSaw script to easily access CZSaw internal objects and APIs, which also enables our CZSaw developers to make quick tests as they develop the system or solve VA problems.

**The General Structure of CZSaw**

The general structure of current CZSaw (v0.1) can be seen in Figure 2.2. The database element uses the MySQL database management system to store text records, entities, and the entity to record relations. Visualization views visualize data and provide the interface for analysts to manipulate the data. The visual history (Kadivar, 2011) provides a visual representation of the analysis history which allows the analyst to visually inspect the analysis process.
CZSaw’s views including document view (read content of documents), list view (a list of entities are placed in a list), hybrid view, and semantic zoom view. Semantic zoom view (Dunsmuir, 2011) is a quick document reader. It allows the analyst to examine documents at several levels of detail: overview (many documents as clusters of nodes), entities in the document, and detailed content. The hybrid view is an enhanced graph visualization. Not only simply displaying entities and relations in a node-link graph, it also enables analysts to mix different types of visualization techniques (e.g., list, grouped nodes, bars, temporal and spatial layouts) to design a visualization for different types of data and different proposes. It allows the analyst to quickly access temporal and spatial layouts without losing the relationships to non-temporal and non-spatial data. The level of data detail can be adjusted to drill down into a dataset without losing the context of the investigation (Chen et al., 2010a). In the following sections and remaining chapters, I will mainly use screenshots generated from the hybrid view.

This system structure shows the data flow from the user’s interaction to the system’s reaction. In CZSaw, the analyst interacts within visualization views. These interactions are captured and translated into transactions (blocks of script statements) of CZSaw script. The script is then parsed by CZSaw to update the dependency graph. These statements may create new nodes and links or update values of the old nodes. CZSaw then propagates through the graph to update all related nodes, which will update visualizations (also nodes in the model) in views. Some executed script statements generate constraint expressions in nodes. During propagation, these constraint expressions are evaluated, which query data, manipulate data, and update visualizations. Updated visualizations and views are captured into screenshots and saved into the visual history. The analyst can interact within the visual history and drive the script to replay to a certain transaction.

Objects in CZSaw

CZSaw was initially inspired by Jigsaw (Stasko et al., 2007) and designed to deal with similar types of data - unstructured text records or documents. These text records are preprocessed to extract entities. An entity could be a person, location, date time, organization, etc. In essence, a text record can be seen as a collection of entities. The text record also defines relations among entities (Chen, 2007). Text records are related to entities due to the direct inclusion relationship (as the record mentioned the entity in its text). Entities
are related if they are mentioned in the same record (co-citation). Records are related if they mention the same entity (bibliographic coupling). Based on such logic, we have three basic types of objects in CZSaw: entity, relation, and record.

**Entity:** An entity in CZSaw has a unique ID, a name, and a value. It could represent a person, a location, date, organization, physical thing, etc.

**Relation:** A relation object defines the relation between entities, an entity and a record, or a record and a record. In the current version, a relation has no direction and no strength.

**Record:** Initially, a record is a text document with entities extracted. CZSaw also accepts rows in a table (a spreadsheet or a table of a relational database) that stores many...
entities as records. For a text document data set (VAST, 2010), the content of a record is important: the analyst has to read the content to realize the real meaning carried by the document and relations among entities (Stasko et al., 2008). But for other data sets that use table rows to store entities and relations (VAST, 2008, 2009), the record is used to define relations among objects.

**View:** CZSaw can have multiple visualization windows. Each window is called a CZSaw view, and has properties such as its location, size, and default layout.

**entityVariable:** When analysts analyze entities, in many cases they need to deal with a set of entities. For example, to find related entities of a given entity (or entities), the result will be a set of entities. Thus, we define an object called the entityVariable to refer to a set of entities. An entityVariable could also deal with a single entity to treat it as a set of a single element.

**relationVariable:** Similar to an entityVariable, a set of relations can be referred by the relationVariable object.

**showNode:** We want to separate the content (entities in an entityVariable) and the display format of the content. If needed, we can then visualize the same content multiple times. For example, the same set of entities can be visualized either in a differing views (windows) or in the same view window, but with a number of differing appearances (such as one node only, as a group node, or a list). We can then use different showNode objects to manipulate these different visualizations.

**showRelations:** Similar to showNode, relations (edges among entities in a node-link graph) may be shown in different formats and in different locations. The showRelation object allows us to refer to one particular visualization of a relationVariable.

CZSaw visualizes entities and relations separately. When used to visualize entities, CZSaw only displays entities (as nodes in GRAPH format or in a LIST) without presenting the connections among these entities. To display connections, the system has to compute relations from the selected entities and display the relations with some layout algorithm.
Functions to Support Analysis on Entities and Collections

CZSaw provides computations for data query, computation, and management. These functions can be accessed through the user interface or via the script view (for advanced control). Script commands include the following:

**Data view commands:** Control visualization states, such as show/hide, layout, and aggregation levels of the visualization (e.g., showing a set of entities as scattered nodes, in a list, or grouped as a single node).

**Data query commands:** Query, filter entities or relations from CZSaw’s database. Examples include searching for entities by value comparison, searching for entities related to a set of entities, or establishing relations between two entity sets. In CZSaw, analysts do not need to deal with a very crowded graph containing all of the entities. They can work selectively on a set of entities.

**Entity refinement commands:** Extract, merge, edit, and link entities. CZSaw relies on entities and relations to generate visualizations. Text documents involved in real-world problems are usually messy and contain inconsistencies and errors. Automated entity extraction is often incomplete. Thus, we provide entity management commands (extract new entities, merge entities, link entity to reports, and unlink entity from reports) that allow analysts to refine entities on the fly within the analysis. The analyst can manually refine entities while reading a document or working within other views (e.g., similar nodes in Hybrid View’s entity network bring about entity-merge possibilities). These entity refinement commands will alter the data in the source database. CZSaw’s dependency propagation mechanism instantly updates content and layout in views to reflect these changes.

### 2.2.2 Example - Identifying Social Network Changes From Phone Call Records

In this section I use the VAST 2008 Mini Challenge 3 (VAST, 2008) to demonstrate the scripting language and interface interactions associated with the script in CZSaw.

The goal of this challenge is to discover the changes in the Catalano social network over the period. The contest provides a data set of cell phone call records over ten days. There are about 400 unique cell phone IDs. These records reveal critical information about
the Catalano social network structure. Each record in the data set has the following values (VAST, 2008):

- From: the calling phone ID
- To: the receiving phone ID
- Date&time: accurate time of the call in the format of yyyymmdh hmmm
- Duration: length of the call in seconds
- Location: location of tower originating the call

To solve this challenge, we first converted the data set into the CZSaw format. There are three types of objects: phone ID, date&time, and tower. Both From and To identifiers are phone IDs. Each row of the records is a CZSaw record. It connects four objects - phone to phone, phone to tower, and phone to date&time.

This challenge asked two questions. The first is to identify the Catalano social network. The second question is to characterize the change in this Catalano social network structure over the 10 day period.

While working on the data in CZSaw, we noticed that some frequently used phone numbers suddenly disappeared in later days. Since the human communication network should be persistent, we suspected that phone numbers which disappeared were replaced by some new numbers. The substitutes should have similar contacts (unless most contacts also changed their numbers) and similar locations (towers) as had the original phones.

The scripts in Figure 2.3 is based on the logic in this solution. Following are the detailed explanation of scripts and corresponding interactions in CZSaw.

**Step One - Search for the suspicious phone**

The first step is to search for and display one phone that needs to be checked. First, let us create a new visualization window with graph layout (Line 1 of Figure 2.3). This line created a new View and assigns this CZSaw object to the variable $gv$. Data visualized in this view is in node-link graph (another option is to display data in a list - “ListView”). In later scripts, the variable $gv$ is used to refer to this data view window.

From previous analysis, phone 5 may be one of the key persons in the social network. Other suspect phones are 200, 1, 2, 3, 97, and 137. Line 2 and 3 of Figure 2.3 search for
CHAPTER 2. THE MOTIVATION: THE PARAMETRIC MODEL IN CZSAW

1. `gv0 = newView( "GraphView" );`
2. `phoneID = "5";`
3. `originalPhone = search(phoneID, "phone");`
4. `showOriginalPhone = show(originalPhone, gv0, "GRAPH");`
5. `relatedPhone = relatedNodes(originalPhone, "phone");`
6. `showRelatedPhone = show(relatedPhone, gv0, "GRAPH");`
7. `relPhoneRel = relations(originalPhone, relatedPhone);`
8. `showRelPhoneRel = showRelations(relPhoneRel, showOriginalPhone, showRelatedPhone);`
9. `replacePhone = commonRelated(relatedPhone, "phone", 3);`
10. `showReplacePhone = show(replacePhone, gv0, "GRAPH");`
11. `relPhoneRep = relations(relatedPhone, replacePhone);`
12. `showRelPhoneRep = showRelations(relPhoneRep, showRelatedPhone, showReplacePhone);`

Figure 2.3: The CZSaw script to search for replacement phone

an entity that with type “phone” and value “5” (or several entities if there are more than one phone entity with value 5), and assign this entity (entities) to a variable `originalPhone` (Figure 2.4). In later scripts, we can use this variable to refer to entities in it.

Line 4 of Figure 2.3 displays entities of `originalPhone` on the graph view window `gv0`. The parameter “GRAPH” tells the view to display entities in the view as distributed nodes (one node per entity). The visualization of `originalPhone` is stored in the variable `showOriginalPhone`. Force-based algorithms (Fruchterman and Reingold, 1991) are used to lay out these nodes by default.

Here, the variable `showOriginalPhone` is a show object that represents the displaying entities and its layout in the view.

Step Two - Find Related Phones

Next, we need to find all related phones to the phone focused in the above search. The analyst interacts with the show object `showOriginalPhone` (the phone 5 node in the view `gv0`) to discover related phones (`relatedPhone`) (Figure 2.5). The corresponding script is line 5 of Figure 2.3.

Script Line 6 displays these phones (`showRelatedPhone`) in the graph view `gv0`.

Line 6 only shows the nodes, but not the edges among them. Line 7 and 8 discover the relations between `originalPhone` and `relatedPhone` and display these relations.
CHAPTER 2. THE MOTIVATION: THE PARAMETRIC MODEL IN CZSAW

Step Three - Detect The Replacement

In the third step, we want to discover if there are any phones that relate to at least three phones in relatedPhone (Figure 2.6). Line 9 queries such phones and assigns it to replacePhone.

Then CZSaw requires several additional lines of script (Line 10, 11, and 12 of Figure 2.3) to display these nodes and their relations.

According to Figure 2.6, there are several phones having three or more connections. Among these phones, phone 306 connects to most phones in relatedPhone. Continuing the investigation by checking the dates of calls, we found that phone 306 was put into use right after phone 5 stopped. Thus, we suspect that it is phone 306 that replaced phone 5.

We can repeat the same procedure to analyze other phones. But certainly the simple repeat of the same procedure is not a good idea. Thus in CZSaw we built a parametric model from the script to let the user reuse the logic contained in the analysis process.

2.2.3 Building the Parametric Model From the Script

Mapping the Scripts to Parametric Model in CZSaw

CZSaw creates a parametric model based on parsing and running the scripts. Running this script may create new nodes and links in the model or update values of existing nodes. In the script, some transactions assign atomic values to or directly use atomic values for a variable, such as the script phoneID = "5", in the above VAST 2008 example. Such script defines the single property source node (the variable originalPhone) in the graph.
CHAPTER 2. THE MOTIVATION: THE PARAMETRIC MODEL IN CZSAW

Figure 2.5: Related phones of phone 5

Figure 2.6: Phones that connect to at least three of the related phones
CHAPTER 2. THE MOTIVATION: THE PARAMETRIC MODEL IN CZSAW

Figure 2.7: The symbolic model created from the script of section 2.2.2

Some lines of script create variables that rely on other variables, which define the constraint expressions of dependent properties. For example,

\[
\text{relatedPhone} = \text{relatedNodes(originalPhone, "phone")};
\]

creates a single property successor node \textit{relatedPhone} with the line of script as its constraint expression, which defines the edge between \textit{originalPhone} and its predecessor nodes \textit{relatedPhone}.

CZSaw visualizes this parametric model directly as a DAG. CZSaw calls it a \textit{dependency graph}. The dependency graph created from Figure 2.3 is Figure 2.7.

In CZSaw’s parametric model, there are several special source nodes, such as the node \texttt{[phdCellphone.xml]} and \texttt{gvo}. \texttt{[phdCellphone.xml]} represents the data source used by this model, and \texttt{gvo} represents a view window. For a static data set, the data source won’t change. Although the analyst can move/resize/hide/show a view window, these operations will not change the content (the graph) within it.

2.3 Reusing the Model

From Figure 2.7, we have a parametric model to search for the replacement phone of phone 5. By replacing “5” with “1” in the variable \textit{phoneID} with the following script, we can achieve the replacement of phone 5 by phone 1:

\[
\text{phoneID} = "1";
\]
As the value of source node phoneID changes, the propagation mechanism evaluates its successors transitively starting with its direct successor node originalPhone. As a consequence, all successor nodes in this model are updated. Hence we get a new graph of phone 1 in the view as Figure 2.8, which suggests phone 309 may replace phone 1.

In this particular example, we need to check 6 phones: phone 1, 2, 3, 5, 97, and 137. In this parametric model, we have other important parameters fixed. One is the object type “phone” in the line:

```
relatedPhone = relatedNodes(originalPhone, "phone");
```

The replacement phone should be active in the same region, which means it should connect to the same towers. Thus, the “phone” could be replaced by “tower” to find all of the related towers to which the original phone connects; this makes the constraint expression of relatedPhone:

```
relatedPhone = relatedNodes(originalPhone, "tower");
```

To search for the replacement phone, we used an integer number “3” in the following script to search for phones that connect to at least 3 phones in relatedPhone:
replacePhone = commonRelated(relatedPhone, "phone", 3);

This number can be considered as a measurement of the confidence level of the replacement phone. The higher the number, the more confidence we have in locating the replacement phone. A higher number could mean more accurate information, for example, all objects in relatedPhone, or at least half of all objects in relatedPhone. Also, a lower number may provide us more useful information about other phones that connect to this social network. Thus, we could replace this number “3” with other possible values, for example, 1, 2, -2 (half of all objects), or -1 (all objects).

By modifying the script or re-assigning values to variables, we can reuse this parametric model to check for other phones and verify the result. Thus we can avoid much repetitive work and enhance the analysis efficiency.

2.3.1 The Problem and the Motivation

Although we could possibly iterate all of these different parameter settings through updating values in the node, it will take a great deal of effort to do so if each parameter has many possible values. In our above cell phone example, there are six phones which we need to examine. For the relational strength, we may want to see a series of numbers from 1 to all phones talk to all phones in relatedPhone. Finally, we may also want to check the active range (towers). As such, the number of final combinations is a multiplication of all parameters, which will be a large number. As a result, it will be quite difficult for the analyst to manage these variations.

2.4 Summary

Derived from the propagation based parametric design systems, the parametric model of CZSaw allows analysts to reuse the solution for an analytical problem.

CZSaw’s script allows the analyst to refine and reuse the recorded analysis process. Editing parameter values in the script and replaying the script in CZSaw helps the analyst to explore alternatives by repeating the whole analysis process with different parameters. However, such an “editing” is a very heavy interaction task for the user; the analyst has to open the script editor, change the parameter, then replay the script.
The propagation mechanism of CZSaw’s parametric model makes this parameter change and alternative exploration easier for the user in general. The parametric model is a constraint solving method, which organizes objects and constraints in a directed acyclic graph. Objects are nodes. Some objects’ (successor nodes) values are constraint expressions that depends on other objects (predecessor nodes), which create links pointing from predecessors to these successors. The propagation mechanism ensures the consistency of the whole model: whenever a node changes its value, all of its downstream nodes (their properties and constraint expressions) are evaluated and populated with proper values. However, once the list of parameters is long, or a parameter has a large number of possible values, the effort to check all possible values is still significant.

The parametric model gives us the basic equipment by which to explore a large number of alternatives by changing the values of inputs (graph independent properties) in the model. A proper way to manage these inputs will give the user the control to manage alternatives of the whole model (each of which is a state of the system). Now the challenge becomes finding a way to let the user effectively organize these inputs and allowing the user to easily navigate through the potentially huge space of alternatives. Thus, I choose to design a formalism to handle alternatives within the parametric model.
Chapter 3

Alternatives in Several Visual Analytics Systems

As an outgrowth of the fields of information visualization and scientific visualization, visual analytics focuses on analytical reasoning facilitated by the interactive visual interfaces (Wong and Thomas, 2004). It helps analysts to make high-quality human judgments with a limited investment of time. Thomas and Cook categorized several analytical tasks that visual analytics tools should support: understanding past and present situations, identifying possible alternative futures, monitoring current events for warning, determining indicators of the intent, and supporting decision making (Thomas and Cook, 2005). Due to lacking the ability to explore and compare a board range of alternatives, many times decision makers make decisions within a limited range (George, 1980). In this chapter, I will review different visual analytics systems to study how analysts explore and compare alternatives within these systems. The exemplary VA systems I have selected to analyze are Tableau (TableauSoftware, 2010), Jigsaw (Kang et al., 2010), VisTrails (Tohline et al., 2009), and GeoTime (Kapler et al., 2005). I also included Adobe Photoshop CS5 (Adobe, 2010a) for the comparison of its capability of supporting graphics design tasks with alternatives.
3.1 Related Guidelines to Better Interacting With Alternatives

George (1980) lists several means by which intelligence analysts and policy makers make poor decisions, including the choice to:

- Select the first answer that appears “good enough”.
- Focus on a narrow range of alternatives, while ignoring the need for a dramatic change from the existing position.
- Opt for an answer that elicits the greatest agreement and support.
- Choose the answer that appears most likely to avoid some previous error or duplicate a previous success.
- Rely on a set of principles that distinguish “good” alternatives from “bad” alternatives.

“Biases and mind sets too often converted subject-matter confidence into arrogance; false assumptions blinded analysts to their target’s true intentions” (Moore, 2007). In his book, Moore (2007) lists many examples of intelligence failure that cause huge consequences. According to Kam (1990), the intelligence analysis is “consistently biased, and … bias is the cornerstone of intelligence failures.” To help analysts overcome these problems, Moore suggests critical thinking as the essential tool for intelligence professionals. Critical thinking “provides part of the solution as it encourages careful consideration of the available evidence, close examination of presuppositions and assumptions, review of the alternate implications of decisions, and finally, discussion of alternative solutions and possibilities” (Moore, 2007). Obviously, dealing with alternatives is crucial in critical thinking.

Thomas and Cook (2005) categorized several analytical tasks that visual analytics tools should support: understanding past and present situations, identifying possible alternative futures, monitoring current events for warning, determining indicators of intent, and supporting decision making. For different visual analytics tasks, the meaning of alternative varies. But whatever it is alternative evidences or alternative solutions, a successful VA tool must be able to provide a way for the analyst to interact with alternatives.
3.1.1 Design Principles to Support Alternatives

Before analyzing the selected systems’ interactions with alternatives, I have chosen three pieces of literature to set up a standard for discussion. Stolte et al. (2002)’s three demands are the basic data-analysis interface requirements; Lunzer and Hornbæk (2008) provide three principles by which to design interface and interactions; and Terry (2005) sets a design criterion for state creation and revisiting. These principles and criterions complement each other and cover from interface design and interaction design.

Stolte et al.’s Data Analysis Demands

Analyzing large multidimensional databases is challenging. In order to support this analysis process, Stolte et al. (2002) have outlined three demands that an analysis tool must meet:

- “Exploratory interface: Analysts must be able to rapidly and incrementally change what data they are viewing and how they are viewing that data as they explore hypotheses.”

- “Multiple display types: Analysis consists of different tasks such as discovering correlations between variables, finding patterns, and locating outliers. An analysis tool must be able to generate displays suited to these disparate tasks.”

- “Data-dense displays: The databases typically contain a large number of records and dimensions. Analysts need to be able to create visualizations that will simultaneously display many dimensions of large subsets of the data” (Stolte et al., 2002).

Lunzer and Hornbaek’s Design Principles of Subjunctive Interfaces

Subjunctive interfaces (Lunzer, 1998) extend applications to support the parallel setup, viewing, and control of alternative scenarios. Through illustrating several system examples from information access, real-time simulation, and document design, Lunzer and Hornbæk (2008) establishes three key design principles for subjunctive interfaces:

- “setting up multiple independent scenarios that exist at the same time”;

- “viewing those scenarios side by side”; and


• “making changes to many scenarios in parallel”.

These principles outline three steps of user interaction with alternatives, i.e., the possibility to explore, interpret, and modify. Although the VA systems discussed in this paper have not purposely developed subjunctive interfaces, some of them do provide the interface where users can view and interact with alternatives.

Terry’s Criterion of State Revisiting

Terry (2005) discussed three conceptually different situations in which a user may find it necessary to explore alternatives:

• “Before a command is invoked. In this situation, the user realizes that the current state of the problem will necessitate exploring a number of alternatives.”

• “While interacting with the command. At this point, users may discover a number of interesting variations, or be unable to find one that perfectly fits the problem.”

• “After a command has been applied. In this case, the results obtained are not as hoped, though not without value. This realization may come immediately after invoking a command, or several steps later, when it becomes more clear that earlier actions must be refined.”

In order to support alternatives, the system should support the user in creating new states, duplicating old states, and switching among states without losing the current state. Being able to navigate, select, compare, and manipulate different states is useful and sometimes critical for data analysis. For this reason, I list state revisiting as another useful criteria by which to assess VA systems.

In the following section, for each of the interactive systems, I introduce the system interface through the lens of Stolte et al. (2002)’s three data analysis demands. Then, I discuss what alternatives are in these systems and how said user explores, interprets, modifies, and revisits such alternatives.
CHAPTER 3. ALTERNATIVES IN SEVERAL VISUAL ANALYTICS SYSTEMS

3.2 Discussion of Five Interactive Systems

3.2.1 VisTrails

Built by researchers from the University of Utah, VisTrails is a visualization management system providing a scientific workflow infrastructure for data exploration and visualization. VisTrails records the detailed provenance of exploratory computational tasks into workflows and uses this information to simplify the process of exploring data through visualization (Sliva et al., 2007). The scientist can easily navigate through the space of dataflows (within one dataflow or across different versions of dataflows) created for a given exploration task. Callahan et al. (2006) emphasized that “it gives scientists the ability to return to previous versions of a dataflow and compare their results.”

Operations in a VisTrails dataflow include visualization operations, application-specific steps (e.g., invoking a simulation script), and general file manipulation functions (e.g., transferring files between servers) (Callahan et al., 2006). VisTrails is a comprehensive analytical tool that contains an exploratory interface, multiple display types, and a data-dense display. The linked graph at the right corner of Figure 3.1 is a VisTrails dataflow.

Alternatives in VisTrails

There are two levels of alternatives in VisTrails. A dataflow generates a series of visualizations based on user defined parameters (the user may assign a range to a parameter) and outputs to the visualization spreadsheet. These visualizations are the first level alternatives, which allow the user to compare different parameter settings (called instances in VisTrails) for one dataflow. The second level of alternatives are the different versions of data flows (the main window in Figure 3.1). Development of these versions makes a version tree. The version tree records the developmental history of each dataflow. Selecting a node in the tree will activate the respect dataflow (at the upper right corner of the main window in Figure 3.1). Lines between versions represent the inheritance relationships of dataflow versions. The user can select and try any dataflow version. The user selects one dataflow (and parameter settings) to produce a visualization. Multiple visualizations are displayed in a visualization spreadsheet to let the user review and compare (visualizations at the right column in Figure 3.1).
CHAPTER 3. ALTERNATIVES IN SEVERAL VISUAL ANALYTICS SYSTEMS

Figure 3.1: VisTrails: different visualizations are generated by different versions of dataflows

Define and Modify: Exploring the Visualization Alternatives

The user works in the pipeline builder window to build a dataflow. He/she starts to explore the visualization possibilities through adding and connecting modules and parameters. After connecting and adjusting the module structure in the dataflow, the user has created a visualization product. In order to explore the variations, he/she can interpolate a parameter value based on specifying the start, end, and step count values. As a result, the visualization product is populated with a series of visualizations (Figure 3.1). In VisTrails, there is a clear separation between a dataflow definition and its instances. A specific set of parameter values comprise an instance. Instantiated with different sets of parameters, VisTrails generates a number of visualizations and allows scientists to explore the parameter space.

Revisit: Exploring the Dataflow Alternatives

The process of creating a dataflow is recorded in a version tree. Each node in the version tree corresponds to one version of the dataflow. A dataflow can be built based on any
version in the tree, which leads to a new branch in the version tree. In VisTrails, a user can execute multiple dataflows (pipelines) by selecting and comparing them within the visualization spreadsheet.

VisTrails is a mature tool by which to visualize and analyze many kinds of data such as complex fluid flows (Tohline et al., 2009) and journal paper formats (Tohline and Santos, 2010). It enables users to revisit their (even others’) past states freely, safely, and productively. Furthermore, it also allows editing on the past states and generating history-based alternatives for the users to explore. Such a two-level structure enables wide exploration.

### 3.2.2 Tableau

Originating from a research project inside Stanford University, Tableau (Heer et al., 2008) is currently a commercial system for visualizing data and relations among relational databases, data cubes, and spreadsheets. Tableau’s foundation formalism, Polaris (Stolte et al., 2002), is based on a specification language called VizQL, in which the statements are generated from user selected data and specify both the data that should be visualized (as database query statements) and how the visualization should appear (as visual specification statements).

Tableau interface (Figure 3.2) includes both a list of available database fields (data dimensions) and a workspace in which users can select fields and drag them onto shelves corresponding to visual encodings such as position, color, shape, and size.

Tableau addresses the analysis demands by providing an interface that generates table-based displays rapidly and incrementally. A table consists of a number of rows, columns, and layers. Each table axis may contain multiple nested dimensions. Each table entry, or pane, contains a set of records that are visually encoded as a set of marks to create a graphic. Stolte et al. (2002) argue that several natural characteristics of tables (such as being multivariate, comparative, and familiar) make them particularly effective for displaying multidimensional data.

#### Alternatives in Tableau

Tableau handles multidimensional data. Datasets, data dimensions, and relations among datasets are all the constructs of Tableau’s alternative space. In Tableau, dimensions and subdimensions of data comprise data alternatives, and different visualization methods
Figure 3.2: Tableau: Compare product sales and gross profit (Heer et al., 2008)
(charts, diagrams, and graphs) provide alternative channels for the user to explore, interpret, modify, and revisit data. Changing data along one (or several) dimension(s) provides alternatives.

Exploring Alternatives

In order to explore data, users start by simply dragging and dropping database fields to the visualization workspace. The user defines the workspace by selecting data dimensions (e.g., category of products, their sales, and profit in Figure 3.2). Tableau automatically generates a series of results (bars) based on the input parameters and these results can be compared side-by-side (the left screenshot of Figure 3.2).

Tableau also uses different visualization methods to show alternative perspectives of the same datasets. Depending on the user’s intent and the character of the data, the user can switch between a bar chart and map for different analysis purposes. Bar charts are good at sorting and filtering values, and maps are effective for checking the data’s spatial distribution. Thus, other than data and data dimensions, visualization methods can be seen as another set of parameters to control the final visualization result.

Interpreting and Modification of Alternatives

In the Tableau workspace, the user can open several different views of the datasets, for example, a color-highlighted table, a side-by-side bar chart, a pie chart, or a map view. To check data changes over time, the user can either examine animations running according to the time dataset or compare visualizations of different time frames side by side.

Users define the dataset and the visualization shows alternatives. They are also free to edit the data in the visualization to see other sets of alternatives. Furthermore, the user can add new dimensions to the scene in order to compare different aspects of the data.

3.2.3 Jigsaw

Developed by the Georgia Institute of Technology, Jigsaw (Stasko et al., 2008) provides an analyst with multiple perspectives on a document collection. Its primary focus is for displaying connections among entities across the documents and providing a type of visual index for the document collection. Jigsaw presents information about documents and entities through multiple visualizations (called views). Each view visualizes a different
perspective of data. In the 2008 version, these views comprise the following: list view, graph view, scatter plot view, document view, calendar view, document cluster view, and shoebox.

**Alternatives in Jigsaw**

The fundamental data types in Jigsaw are documents, entities, and relations. Synthesizing these data types together constructs the visual analytical space. According to an evaluation study conducted on Jigsaw (Kang et al., 2010), most users execute a strategy of “find a clue and follow the trail” while analyzing documents. The clues they find are usually a comparatively strong connection between entities. Believing that this connection will lead to a productive scenario, the user will follow the trail rigorously by using search or other functionalities provided through the use of the tool. In this case, the “clue” and following analysis results build up as a hypothesis. Different hypotheses can be seen as alternatives. Considering the details in the narrative, the individual document/entity and its related objects (relations, documents, and entities) are actually considered to be a lower level of alternatives.

**Exploring Alternatives**

The investigation process in Jigsaw usually starts from a small hint: i.e., somebody or some event is a key entry point into the maze. Such a hint may come from scanning through several documents in order to search for a frequently mentioned entity or from scanning through the list view to check for connections among up to three lists of entities.

It is possible to open and compare two graphs from the same dataset in one Jigsaw application (Figure 3.3). The user can separately control (open or close the event listening) each graph view window to control the synchronization of graph updates. When the user interacts with objects in one graph window to update the graph (e.g., open a document to see all related objects), other windows containing the same object can be set to synchronize the graph to reflect the changes on the same objects or to remain consistent with the old graph.
Modification of Alternatives

It seems Jigsaw has no support for “modify many scenarios in parallel.” In its graph view, the user has to manually click on individual entities one at a time. He/she has to repeat similar steps of interactions when working on multiple alternatives. There is no way to either reuse previous actions or edit the data involved in previous actions to update the graph for new data.

Revisiting Alternatives

Jigsaw can bookmark the current investigative state. Through saving many bookmarks, it is possible to revisit one of the past states. However, there are several drawbacks I found while experimenting with the system. First, the user has to bookmark the current state before retrieving previous bookmarks, or the current state will be lost. Second, there is no connection information among bookmarks by which to tell the user how one bookmarked state was developed. Third, bookmarks made in either view cannot be transferred or shared in another view. Fourth, while we can open two graph views as an ad-hoc way in users to assist comparing two graphs, the method is fragile.
3.2.4 GeoTime

Developed by Oculus Info Inc., GeoTime (Kapler et al., 2005; Proulx et al., 2006; Kapler et al., 2008) is a visual analytics tool to display and work with data over both space and time within a single, highly interactive 3D view. Viewing events, connections, and movements in a combined temporal and spatial view allows analysts to detect, examine, and understand patterns quickly.

Major object types employed in GeoTime are entities (people or things), locations, and events (occurrences or discovered facts). Events are describable actions (for example, Sam called Liz on Wednesday). Events store the times at which the action took place. These objects are organized as associations, which describe pairings between objects, such as event A occurred at location B, or entity X was present at event. GeoTime defines groups of associated elements to represent certain classes of occurrences and relationships. For example, the group of “Observation with actor” groups two associations: an event occurred at location A, and entity X present at the event. These groups all have specific visual expressions and interactive behaviors in the display (Kapler and Wright, 2005).

GeoTime provides insight into events and behaviors in time and space. GeoTime represents events within an x-, y-, t-coordinate space, in which the x-, y-plane shows geographic or diagrammatic space and the vertical t-axis represents time. Events are plotted in the three-dimensional view. Analysts can then visually recognize and interpret closely related events in time and space. GeoTime has a main 3D visualization view and a small calendar view on the bottom. It surmounts the apparent problem of providing only a single display type by its typical use in conjunction with another Oculus product, nSpace, which provides multiple display types.

Alternatives in GeoTime

Because of its 3D visualization, GeoTime is able to display many entities, locations, and events together in one view. Plots of events can be seen as alternatives in the analytical space. Associations among objects, as well as each individual object (entities, events, locations), are alternatives in a detailed level. For example, Figure 3.4 visualizes taxi movement data in a scene of a hit and run accident. Taxis’ traces can be as alternatives to each other.
CHAPTER 3. ALTERNATIVES IN SEVERAL VISUAL ANALYTICS SYSTEMS

With GeoTime, through importing multiple datasets in different layers, the user combines datasets, filters information, and detects commonalities across datasets through the visualization. This allows the user to examine the same dataset from multiple angles, which is useful in discovering unexpected patterns (Chien et al., 2007).

**Exploring, Interpreting, and Manipulating data**

GeoTime displays multiple plots and objects in the same view. Exploration, interpretation, and manipulation of alternatives are integrated. GeoTime’s interactions include filtering, zoom-box selection, saved views, and simultaneous spatial and temporal zooming, which allows the user to quickly move to a context of interest.

GeoTime is an alternative exploration tool in cases directly related to time and space. Through sliding the timeline bar (the left vertical bar in the screen), GeoTime animates the position of taxis on the 2D map. The user can also rotate the 3-D view to examine the plot from different angles. Figure 3.4 compares the same plots but within different time periods. The left one is a zoomed-in mode to display the plots in the selected time range, checking the location of taxis in detail. The user can add other related events, such as pickup and drop-off, into the scene to gather more evidence.

Thus, the user can change the visualization by adjusting many parameters. Some parameters, such as the dataset and selected objects, define the data to be visualized in the 3-D view. Rotation, view angle, and time range define the appearance of the visualization.

GeoTime can save views and export the view to files for later import. At the basic
CHAPTER 3. ALTERNATIVES IN SEVERAL VISUAL ANALYTICS SYSTEMS

level, the user can revisit a previous view (an alternative representation of data). However, GeoTime provides no direct tools for the user to compare two views.

3.2.5 Photoshop

Originating from an image editing program developed by Thomas Knoll in 1988 (Schewe, 2005), Adobe Photoshop (Adobe, 2010a) is currently the most popular image editing program in the world.

Alternatives in Photoshop

Photoshop allows users to manipulate an image by adjusting its brightness/contrast, hue/saturation, and color balances or by applying special effects. Such image manipulation work is a simple design task. It is open-ended, with no objective standard, and highly relies on the user’s judgement. The possible variations of the image caused by these image adjustments define a large variation space.

Photoshop is a tool for editing images, which contribute to communication material, such as websites, brochures, banners, etc. Being able to choose from alternatives is important for such uses. First, the designer may want to explore and compare different design ideas and solutions (such as a color schema or a layout). Second, many design tasks require a series of designs based on the same schema but with different components. For example, a website design may require several layouts for both the homepage and different sections. These variations may share the same top banner, color schema, and background, but differ in content layout. Third, clients normally want the designer to provide several variations (e.g., in different color or layout) to allow them to compare choices and to select a preferred design. Photoshop provides several means for users to design alternatives.

Exploring Alternatives

Alternatives are generated while a user is manipulating an image. To adjust an under exposed photo, the user may simply try adjusting the brightness and contrast (other better methods may exist) until he/she feels the photo looks right on the monitor. During this process, many alternatives are generated and replaced. In Photoshop, almost all image adjustment operations provide a “preview” designed to allow the user to instantly see the
effects of parameters without changing the original document. Photoshop also provides a better way for the user to see and compare variations (Figure 3.5). The Variations command allows the user to adjust the color balance, contrast, and saturation of an image by showing thumbnails of variations. This command does not require the user to specify the value of the adjusting parameters. It is most useful for images that do not require precise adjustments (Adobe, 2010b).

As a graphic design tool, Photoshop provides painting tools, such as brush and pencil tools. The user can directly select the shape and size of the tool and can then pick the most suitable combination for the task at hand. Furthermore, the newest CS5 version allows the user to see the shape change of the brush during the painting process, assisting the user to understand how much more he/she can press.

In professional practice, designers use layers to accommodate design alternatives (Figure 3.6). Photoshop layers are like sheets of stacked acetate. The designer can see through transparent areas of a layer to the layers below. Designers use layers to perform tasks such
CHAPTER 3. ALTERNATIVES IN SEVERAL VISUAL ANALYTICS SYSTEMS

Figure 3.6: Photoshop: Use of combinations of layers to retrieve alternatives. Courtesy York & Chapel, Inc.
as composing multiple images, adding text to an image, or adding vector graphic shapes (Adobe, 2010b). Through careful arrangement, the design document can be organized in multiple layers, and each layer only contains some elements of the design. For example, in a website design, the designer may use one layer at the bottom as the background, one layer on top for the company logo in the header area, and use other layers in the middle to handle the page layout design of contents. The designer can freely combine layers by turning on/off certain layers, thus displaying alternatives. Figure 3.6 shows a website design (Courtesy York & Chapel, Inc.). The design has two choices of background, two choices for the top navigation bar and banner, and several layout layers for product details. Through turning on certain layers, the designer is able to present to the customer many choices without any tedious duplication work.

Interpreting Alternatives

The variation function allows the user to compare several alternatives by displaying several variation thumbnails. Some of the command’s preview functions also provide small windows to allow users to preview the adjustments and compare these with the original. Photoshop allows the user to duplicate the working document in several ways. The duplicated document is a new document with its own working space. The user can freely arrange these windows to compare duplicated instances.

Modifying Alternatives

When a user employs a combination of layers to generate alternatives, some layers are common layers that exist in many alternatives (e.g., backgrounds, header portions of website designs, etc.). Adjusting these layers will affect several (maybe all) design alternatives. The user may have a set of such layers, which will generate a set of designs. Instead of directly adjusting the pixel values of the image, a user can create an “adjustment layer” to tune the image. An adjustment layer applies color and tonal adjustments to an image without permanently changing the pixel values. Adjustment parameters are stored in the adjustment layer and apply to all the layers below it. The user can change multiple layers by making a single adjustment, rather than adjusting every layer separately (Adobe, 2010b).
Revisiting Alternatives

Photoshop has a history-tracking mechanism to permit the user to track actions taken in the design process, as well as to manage snapshots of design states. There are three levels of revisiting in Photoshop. First, at the system’s level, a user can use undo and redo operations to go back to the previous state of the document. Second, the user can create a snapshot to record the current state of the document within the history panel (Figure 3.7). Third, the history panel automatically records a series of actions taking place in the design process. The user can go back to any of the previous actions and replay actions. However, if the user issues a new command after an undo operation, the undone actions will be lost and replaced by the new command. The user can use the system-level undo to return the document to the previous states and to retrieve the old history actions. At the snapshot level, the user can take many snapshots to record alternatives. The user can freely switch between snapshots. Snapshots are listed in a temporal order in the history.
panel. Sometimes the user may continue the design based on one of previous snapshots. However, the new snapshot is always located at the bottom of the snapshot list. As a result, often the user cannot see the implicit inherited relations that exist among these snapshots.

Other than rolling the whole image back to a previous state, Photoshop also provides a history brush tool for the user to reveal a previous state of painting an image in a free form way. Brush painted areas will be rolled back to a previous state (or can be seen as the user paints on the image using a previous state). The to-be revealed previous state can be selected from the history palette. Thus the user can selectively and partially visit history.

History information will be lost if the user closes the document. History information cannot be saved with the file. Alternatives stored in layers will be kept and can be reused at any time. But in order to retrieve alternatives, the user has to manually combine layers after opening the document.

In Photoshop, undo/redo, snapshots, and actions history are all interwoven together. Photoshop’s history tracking mechanism provides users with the freedom to navigate through design alternatives and the states of alternatives, but utilizing these features in an effective way relies highly on the user’s expertise. Using Photoshop for alternatives is more a matter of idiom of skill that of good system support.

3.3 Summary

Terry et al. (Terry et al., 2004) divided current mechanisms and methods for working with variations into four categories: prototyping and sketching tools that encourage the rapid solution production, tools that track snapshots of the entire document, ad-hoc user strategies for embedding alternatives directly within the document, and enhanced preview mechanisms (“what-if” tools) that provide broader potential future views. According to this division, we can group the interactive capabilities of the five systems we reviewed as follows:

- Prototyping and sketch:
  - In VisTrails, the user can try any flow/parameter in a dataflow, or create a variety of dataflows quickly to sketch the visualizations. Linkages are built up
when input and output values of modules are connected.

– In Tableau, the user can drag and drop datasets to the table columns and rows to generate a chart or diagram to explore.

– In Jigsaw’s graph view, the user can freely explore any entity/report and follow links to test out a hypothesis.

– In Photoshop, layers and adjustment layers provide some freedom for the designer to test out ideas without worrying about permanently destroying the overall design.

– In GeoTime, the user can freely adjust the viewpoint, or zoom in/out the time period without changing the data.

• Track snapshots:
  – VisTrails’s version tree
  – Tableau’s history interface (TableauSoftware, 2010)
  – Jigsaw’s bookmark functions in the graph view
  – GeoTime’s save view function
  – Adobe Photoshop’s snapshot in history panel

• Ad-hoc user strategies:
  – In Photoshop, designers combine layers to create and edit variations.

• Enhanced preview: Both VisTrails and Photoshop provide preview options when the user is editing parameters within the system. However, the other three systems only allow the user to return back to the previous state by restoring the relevant parameter or action.

Lunzer and Hornbæk (2008) outline three subjunctive interface design principles. Although the five systems reviewed are not purposely designed to serve for parallel analysis and design, some of their characteristics meet the subjunctive principles. Here, I group the systems into Lunzer and Hornbæk’s structure:

• Setting up multiple independent scenarios: For all of the five systems, it is possible to set up multiple alternatives at some level within the session, although for some of these systems (e.g., Jigsaw) the user has to use an ad-hoc way to accomplish this.
• Viewing those scenarios side by side: Tableau and VisTrails support this principle well by using the spreadsheet to place visualizations into cells. Jigsaw and GeoTime can also represent many alternatives into the same view. In Photoshop the user can duplicate the document to compare variations or can use limited tools such as variations.

• Making parallel changes to many scenarios: In GeoTime, users’ operations on the 3D view will automatically affect all plots. In Photoshop, users can update a common layer or use an adjustment layer to change all designs. Changes made in VisTrails and Tableau can be applied to multiple instances. But in Jigsaw, the manipulation of several alternatives at once is very difficult. It seems that the user has to manually go through each alternative on a one-by-one basis.

The four visual analytics systems focus on different types of datasets, and solve different kinds of problems. They all require the user to begin exploration in a large space in order to make sense of the available data. The meanings of alternatives are different in these systems. Naturally, visual analytics tasks involve many kinds of alternatives. The analyst has to set up alternatives, examine and compare alternatives, and then draw some insights from this data. The characteristics of the data and alternatives vary in different tasks, so the way the user interacts with alternatives also varies. In some systems the alternatives are very explicit (e.g., VisTrails), while alternatives in other system may be a little bit blurred (e.g., Jigsaw). Although Jigsaw provides abundant perspectives for the users to read and analyze data, it is not sophisticated enough to handle updating alternatives and does not give the user enough freedom to navigate among alternatives. Clearly the analyst (or the designer in Photoshop) may benefit from having many alternatives to explore and interpret. A system with effective functions for the navigation and manipulation of alternatives will support its users and their tasks in a better way.
Chapter 4

Representations of Alternatives

Users explore, compare, and manipulate alternatives. In this section, I select three different domains (software configuration management, product family design, and design space exploration), taking an example system from each to discuss the representations of alternatives. In each system, I organize the content as follows:

- what is an alternative and why alternatives are important for the system,
- representations of alternatives,
- operations on alternatives, and
- a summary of the alternatives and the underlying representation.

4.1 Directed Acyclic Graph to Represent Software Configuration Management

4.1.1 Alternatives: Branches in Software Development

Alternatives in software development can be seen as different versions and different lines (branches) of the development process. Software development may involve many developers working together. Each developer works on his own computer with his own working code focusing on a portion (or feature) of the system. Before constructing the end product, there will be many active simultaneously tasks. Also, other than the “main”
build, designers/developers may want to attempt some new ideas or want to make different versions for different purposes (e.g., the Home edition, Professional edition, Server edition of the Windows system to fit different markets). Managing these alternative versions is essential in software development.

Git (Torvalds and Hamano, 2010a) is a distributed SCM (Software Configuration Management) system for large projects. SCM is a methodology to control and manage a software development process. It is a “set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made” (Pressman, 2004). Git’s internal data structure, as well as its manager of handling branches and history, can be represented and visualized as a Directed Acyclic graph (Figure 4.1). Some of its key characteristics are (Baudis, 2009):

- Strong support for non-linear development. Git supports branching and merging, including tools for visualizing and navigating a non-linear development history.

- Distributed development. Each developer maintains a local copy of the entire development history.

- Cryptographic authentication of history. The Git history is stored in a way that the name of a particular revision (a “commit” in Git terms) depends upon the complete development history leading up to that commit.

Git’s built-in model allows the user to manage multiple local branches that can be entirely independent of each other (Figure 4.2). Git encourages a workflow that branches and merges frequently, even multiple times in a day (Chacon, 2009). With such a branching model, the developer can:

- Try out new ideas by creating a new branch without losing the original work.

- Keep one branch for production. Use other branches for testing and day-to-day work.

- Once a new idea is proofed, the developer can merge it back in. Or he/she can simply abandon it if the new idea is not working.
4.1.2 Representation of Branches

Git Structure and Objects

A Git user may start to work on a local repository by cloning a remote repository. He may also set up a local repository. This repository will become a remote repository for other users. Git maintains a staging area (called an index) and an object database (the repository) in the local computer. The user communicates with remote repositories through git-push (update remote repository) and git-fetch (update local repository). The local repository organizes and holds all objects. The developer works in a local working directory. The index is an intermediate layer, connecting the object database and the working directory. The user checkouts files or merges changes from the local repository into the working directory, and adds new or changed files from the working directory into the index, and commits files to the local repository when they are ready. The index is a preparation area for the full version. The user adds, removes, and renames files into the index, and then the user commits the revision to the repository. While checking out, the index reads the tree from the repository and then populates the local working directory.

There are four types of objects in Git’s object database - Blob, Tree, Commit, and Tag (Torvalds and Hamano, 2010b; Virtanen, 2010) (Figure 4.1). Each object is named by a unique 40-digit SHA-1 value, which is generated by using the SHA-1 hash of the contents of the object. The same content in different locations is always stored under the same name. The cryptographic function of SHA-1 makes it impossible to locate any two objects with differing content under the same name. Instead, Git can quickly distinguish objects by comparing their names and detect errors by checking the consistency of an object’s SHA-1 name with its contents.

**Blob object**: a binary blob of data. The content of a file is a blob object. The same SHA-1 name means that the contents of objects are the same.

**Tree object**: a tree object organizes one or more “blob” objects and/or other tree objects into a hierarchical directory structure. In essence, it represents the contents (files and sub-directories) as a directory tree.

**Commit object**: contains the information about a particular revision. It contains the tree object that designates the top directory hierarchy at the time of the commit. More importantly, a commit has reference to its “parent” commit objects (immediately
previous commit objects which lead to the commit), which creates a directed acyclic graph (DAG) of revisions (Figure 4.1). This DAG describes the history of how the user has arrived at the current state. The initial revision of the projects is represented by a "root" commit. A project may have multiple roots in some rare conditions.

**Tag object**: symbolically identifies and signs other objects. It contains an object, object type, tag name, name of the tagger, and a message.

**Symbolic reference**: Git also provides “references” or “refs” as pointers to help allocate commits without keeping the SHA-1 name. A symbolic reference called “head” is used in Git to indicate the current working branch.

The right image in Figure 4.1 is the completed directed acyclic graph. In this figure, object names such as “33a3fo” and “6f0104” are the first six characters of the 40-digit long SHA-1 unique object name. The commit object “33a3fo” is a direct successor of commit object “6f0104”, while commit “6f0104” is the parent of commit “33a3fo”. The blob object “9daeeaf” shows no changes in these two commits and is shared in two trees. The “Head” and “Branch Name” nodes are references. The branch name gives the user an easy and meaningful way to refer to a branch instead of using the 40-digit SHA-1 name. “Head” indicates the current working branch.

**History and Branches in Git**

Every Git commit object represents a state of development and associates with a compressed snapshot of a file hierarchy. This makes the repository a tool to manage a non-linear history of a collection of files by maintaining the relationship among commit objects. Since all nodes are laid out according to the successor relationship, it is naturally a visualization of the development history (Figure 4.2).

In Git, the same diagram like Figure 4.2, represents both the development history and branches. In this diagram, time goes from left to right (arrows point to the children on the right). Nodes are revisions. I use yellow nodes to represent revisions of the master branch. Green and blue nodes are revisions for branches for testing new functions and ideas.

A commit object itself contains who made the latest change, what they did, when, and why. Parents of commit show what happened before this commit. Following the chain of parents will eventually lead to the project root, which chain retrieves the full history.
Figure 4.1: Git: object hierarchy and linkages
Figure 4.2: Git Workflow. Each node represents a revision. Edges point from the parents to the children.

of a file. Furthermore, Git allows lines of development (branches) to diverge and then re-converge. The point where two lines of development re-converge is called a “merge”. In such a case, there will be more than one parent for the commit object created by a merge. Each parent represents the most recent commit on one of the lines of development leading to that point.

As a distributed system, a remote developer may have his own branches. The local user can selectively share only one branch by keeping other branches private. He is then free to work on the new ideas without disturbing the progress of others.

Operations on Alternatives: Working with Branches

Creating a new branch in Git is simple. The command “checkout” will automatically create a new branch which starts at the current commit. In Figure 4.2, revisions $C_2$ and $C_3$ are both branches checked out from $C_1$. Git has two fundamental operations on branches (Torvalds et al., 2010):

- Git-merge: join two or more development histories together.

- Git-rebase: forward-port local commits to the updated upstream head.

Merge

In the diagram of Figure 4.2, revision $C_5$ is a merge result from $C_3$ and $C_4$. To do a merge, Git automatically locates the common ancestor $C_1$ of $C_3$ and $C_4$, compares the snapshots
The development is split into two branches

Merge two branches

Rebase to copy the change from one branch onto the other branch

Figure 4.3: Merge vs. Rebase. C₅ has the same content as C₄′.

(directory tree and files) in these three commits, generates a new snapshot, and creates a new merge commit.

Rebase

With Git’s rebase command, the user can capture all the changes committed in one branch and reapply them to another branch. In the example demonstrated in Figure 4.3, the content inside C₅ (from merge) is exactly the same as the content inside C₄′. The version C₂′ is generated from C₃ by adding the difference of C₂ to C₁. Then the difference of C₄ to C₂ is added to C₂′ to generate C₄′. These two distinctive operations lead to the same outcome.

Compared with merging, rebasing results in a simpler DAG by removing the branch into a single line. Rebasings replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.
CHAPTER 4. REPRESENTATIONS OF ALTERNATIVES

Summary of Alternative Representation in Git

Git uses the directed acyclic graph (DAG) as the representation (internal and external) of the software development process. A commit node represents a state of development (revision), and the directed edges among commit nodes shows the development sequence. Such a representation synthesizes alternatives (branches) and history. A branch (and nodes on the branch) is an alternative of the software, and the sequence of connections shows the development history. Git visualizes this graph directly to support individual developers as well as a team to manage many ideas and concurrent development. Git’s merge and rebase functions provide users with the ability to manipulate branches.

The history/branch graph does not have the propagation capability. Once a commit has been made, it becomes a part of the history. People can remove/replace/rebase this commit, but the content of this particular commit can never be changed. The user can see how the history and branches are constructed and can further develop branches, but cannot re-use this structure.

Applications from the software family are based on common components, but they tend to serve different user groups and have different features. We can consider them as many branches growing from the same trunk. The trunk part represents the basic shared components. Ideally, when the trunk is updated with new features or bug fixes, all of its branches should inherit the updates automatically. The DAG graph illustrates all of the necessary information about the inheritance of each final version (branches) moving towards the common trunk. A simple propagation along the path should keep every branch up-to-date. However, Git treats the common part (trunk) exactly like a branch. All versions have to be manually merged with the common branch every time there is a new update. Because the current Git structure synthesizes history states with branches and uses the cryptographically SHA-1 naming strategy, there is no quick and easy way to add a propagation mechanism to the graph.
4.2 Typed Feature Structures to Represent Design Space Exploration

Many design problems are complex, intricate problems that have no single correct answer. These ill-defined problems lack clearly defined goals and solution methods (Reitman, 1965). Since there is no right answer, a solution can always be improved on (Rittel and Webber, 1984) within the availability of resources (e.g., time, budget). The lack of well-defined goals means that the goals are flexible and subject to human reinterpretation. Thus, the solutions of design lead to a space of possibilities for individuals to consider. One way of supporting design is to involve computers to help humans find or generate alternative design solutions and to compare them. Design space exploration is the idea that computers can usefully augment design as the act of exploring alternatives. This involves representing many designs, arranging these designs in a network structure (design space), and exploring this space by traversing paths to find new design solutions (Woodbury and Burrow, 2006). Design space explorers use computation to support design space exploration. Woodbury et al. (Chang, 1999; Burrow, 2006; Woodbury et al., 1999) proposed using typed feature structures (TFS) as the representation of such a design space and shaping the action of design space explorers (Figure 4.4).

4.2.1 Alternatives: Design Solutions

In the domain of design space exploration, design states are the alternatives. Woodbury et al. (1999) proposed the mapping from building design domain to the typed feature structure. A building design can be represented by three set of components, types, structures, and descriptions. Types stands for expressed knowledge of building design, structures represents building and components, and descriptions are utterances in a formal textual language. Typed feature structure provides following aspects to a design space explorer (Woodbury et al., 1999):

- structures are both intensional and partial
- partialness is in the eye of the beholder
- a generative system without rules
- an exploration system
CHAPTER 4. REPRESENTATIONS OF ALTERNATIVES

Figure 4.4: Typed Feature Structure: A single-fronted cottage house, its plan, and partially represented feature structure (Woodbury et al., 1999, 2000)

- monotonic generation
- design spaces are structured by subsumption
- reuse is intrinsic

In order to demonstrate this idea, Woodbury et al. (1999) used a single-fronted cottage house design example. A series of design solutions for the cottage house were the design alternatives.

Representation: Typed Feature Structures

Feature structures have been used primarily in the area of computational linguistics and logic programming. They are an essential part of many linguistic formalisms and the underlying representation for language engineering applications. Terms, representations, and definitions have been defined in a draft ISO standard ISO/CD 24610-1 (ISO2004, 2004).

Feature structures are general-purposed data structures that identify and group individual features together by assigning a particular value to each feature. A value can be an
atom value (e.g., integer, string) or a complex value as a feature structure or a collection of values (e.g., set). There are two intuitive views of feature structures (ISO2004, 2004):

- A set of feature specifications that consist of pairs of features and their values.
- A Labeled directed graph with a single root where each arc is labeled with the name of a feature and directed to its value.

In a particular domain, elements can be sorted into classes called types in a systematic way, based on the common properties amongst them. Typed feature structures (Carpenter, 1992) use types to organize feature structures into natural classes. The value of each feature has a type. Types can be organized in an inheritance hierarchy based on their generality. Typing provides a constraint on the content of a feature structure. Two feature structures can only be unified if their types are compatible according to the primitive hierarchy of types.

A typed feature structure can be modeled as a directed (possible cyclic) graph with labels on every edge and node (the right graph in Figure 4.4). This graph notation contains a single root, labeled and direct arcs, and nodes. Each node is labeled with a symbol representing its type (e.g., sf_house, massing), and the arcs are labeled with symbols representing features (e.g., KITCHEN, SLEEPING in Figure 4.4). The type can either be an atom or a complex object that is a feature structure itself. In the graph notation, we can easily see many paths. A path (denoted \( \pi \)) is a sequence of labeled arcs connecting nodes in the graph. Each path starts with the root and goes through each branch, until it reaches a terminal node. Typed feature structures can represent partial information about an object by specifying parts of all feature-value pairs. It is possible to model the known portion of an object at some intermediate stage with general types and parts of the features until more specific information becomes available.

Other than types and feature structures, descriptions is another key component in typed feature structures. Descriptions are textual objects, which call out feature structures through the satisfaction relation. If a feature structure satisfies a description, then so does every feature structure which it subsumes (Carpenter, 1992). In a design space explorer, descriptions act as textual specifications of functional decomposition (Woodbury et al., 2000).
4.2.2 Operations of Typed Feature Structure

Measuring Feature Structures: Subsumption

Feature structures are particularly important for capturing and representing partial information. Subsumption is the notion for comparing feature structures in order to specify which structure carries more or less information. For example, the feature structure $A$ subsumes the feature structure $B$ (written $A \sqsubseteq B$), if $A$ is either less than or equally informative as $B$. To decide feature structures subsumption $FS_1 \sqsubseteq FS_2$, the following conditions must be satisfied:

- If a feature $f$ is defined in $FS_1$ then $f$ is also defined in $FS_2$ such that the value in $FS_1$ subsumes the value in $FS_2$.
- Path equivalences: Every two paths which are shared in $FS_1$ are also shared in $FS_2$.
- Type ordering: Every type assigned to a path by $FS_1$ subsumes the type assigned to the same path in $FS_2$ in the type ordering.

In the context of design, as the design becomes more detailed during the process, the earlier design then subsumes the latter, more detailed ones.

Unification

Unification and generalization provide ways to compute feature structures. Unification combines the amount of information represented in the feature structures into a new feature structure. The new feature structure will be subsumed by the original feature structures. Formally, the unification of two typed feature structures $F$ and $F'$ is the least upper bound of $F$ and $F'$ in the collection of typed feature structures ordered by subsumption (Carpenter, 1992). By contrast, generalization finds identical feature specifications and puts them into one general feature structure. Unification provides a disciplined way to combine information or to add information to a structure. In a design context, the result of unification produces a new, more complete object that is consistent with the objects represented by the argument feature structures (Woodbury et al., 1999).
Exploring Alternatives: \( \Pi \)-Resolution

Burrow and Woodbury demonstrated design space exploration to discover alternative designs by using incremental \( \pi \) resolution as an interactive constraint resolution mechanism. The implicit space contains states that are made possible by the generative mechanism. The explicit space comprises those states that have been visited during the design process (Woodbury and Burrow, 2006). A type hierarchy defines an implicit design space, and the \( \pi \)-resolution provides incremental steps for the user to reveal the explicit space.

\( \pi \)-resolution is a feature structure constructing process. It searches across a sequence of subsumed feature structures \( F_1 \sqsubseteq F_1 \sqsubseteq F_2 \ldots F_k \) for a feature structure \( F \) that satisfies the constraints on each substructure. The initial feature structure in each sequence is a most general satisfier of the query description. \( \pi \)-resolution is an incremental searching strategy. It solves type constraints one substructure at a time. A complete search strategy can be ensured by a breadth-first strategy that selects the unresolved path of minimal length (Carpenter, 1992).

Incremental \( \pi \)-resolution allows the explorer and the designer to interactively work together. In design space exploration, the designer decides what the next operation in the space should be and takes local control over the paths. Design decisions (decisions upon alternatives) are not subject to some global inference strategy, but are goal-directed. Each resolution step introduces new constraints and opens up possible spaces via the designer-explorer interaction (Woodbury et al., 1999).

4.2.3 Summary of Typed Feature Structure

Typed feature structures provide a faithful representation for design by supporting its partialness, structure sharing, intentionality, and cyclicity. Its well-founded theory, logic, and operations provide a feasible way for the designer to be involved in computers in exploring the hidden and vast design space, which contains all possible design states.

In Penn’s 2006 paper response to Woodbury (Woodbury and Burrow, 2006), he pointed out that TFS have some weaknesses in the context of natural language grammar design, especially in terms of large grammar design (Penn, 2006). Such issues may arise for large-scale design tasks as well.
4.3 And/Xor Tree to Represent Product Family Design

4.3.1 Alternatives: Customizations in the Product Family Design

Customers want more choices; they want to find a product which fits their own preferences. The successful story of DELL computer reflects such a need in the markets. A user goes to the website http://www.dell.com and chooses a hardware/software configuration fit specifically for his or her needs with nothing more and nothing less. Mass Customization Manufacturing (MCM) (Pine, 1993), which aims at satisfying the individual customer’s need, has been widely adopted by the industry. However, the number of variations is, to a large extent, based on the degree of customization freedom. A wide range of combinations may result in millions of variations. One effective way to deal with such MCM problems is Product Family Design (PFD) (Jiao and Tseng, 1999c). As defined by Meyer and Utterback (1993), a product family is formed by products that share a common product platform but have specific features and functionalities required by a different set of customers, while a product platform encompasses the design and components shared by a set of products. The product platform can work for a series of closely related products. Product family design can be modeled as configuration design, which is mainly concerned with determining which components/modules among a pre-defined set are in a design and how they are arranged spatially and logically to satisfy customer requirements and engineering/physical constraints (Mittal and Frayman, 1989). The set of components used in configuration are pre-defined. No new component can be added into the problem. The set of requirements and constraints is assumed to be complete. The hierarchy of partial assemblies are also known beforehand. Thus, the alternative space has been defined with clear boundaries. PFD essentially entails a configuration problem by “combination” to generate a family of configuration design alternatives simultaneously within a single design activity.

4.3.2 Representation: And/Xor Tree

Jiao et al. (2000) proposed a Product Family Architecture (PFA) generic variety structure (GVS) representation (Figure 4.5): an and/xor tree. Users can understand this generic structure from three perspectives: product structure, variety parameters, and constraints.

**Product structure:** All product variants in one family share a common structure. It can be
Figure 4.5: AND/XOR tree: a generic structure for characterizing variety (Jiao et al., 2000, 2007)
described as a hierarchy comprising constituent items \((I_0, I_1 \ldots)\) at different levels of abstraction, where \(I_i\) nodes can be either abstract or physical entities. The topology of the end-product configuration can be seen from such a breakdown structure \((\text{and}, \text{tree})\). Product families are distinguished by the common product structure, which is represented as the set of \(I_i\) and their interrelations within the graph. The depth of the tree depends on the product structure, which may have more than one level of sub-modules. In the graph, constituent entities can be used to represent different aspects of a product family, such as functional features, technical parameters, or components/assemblies.

Variety parameters: The item \((\text{a node } I_i)\) may have attributes that are relevant to variety and thus are defined as variety parameters \((P_j)\) in the graph. Each parameter may have several instances from which to choose. The vector of variety parameters, in this graph \(P_{11}, P_{11}, \ldots, P_{32}\), represents the full specifications of the product family. One particular product variant within this family is an instance of the \(P\) vector, such as, \(\{V_{11}, V_{12}, V_{21}, \ldots V_{31}\}\).

Configuration constraints: There are two types of constraints among these items and parameters. A product family can be viewed from differing perspectives, such as the specific view of the functional, behavioral, or physical (Jiao and Tseng, 1999b). Within one particular view, restrictions on the combination of parameter values are categorized as Type I constraints. Given a chosen value in some variety parameters, the choice of other variety parameters is limited. Constraints across views are referred to as Type II constraints. Most of Type II constraints are about design knowledge. They can be described as rules instead of being graphically depicted in the generic structure (Jiao and Tseng, 1999b).

According to the customizable features represented in the generic variety structure, a configuration space (Figure 4.6) can be developed. This configuration space includes all possible configuration alternatives. Consistent with the GVS, the configuration space is established as a hierarchical structure. Feasible configuration design alternatives, modules, features, design parameters, and their relationships are described in a single formalism. This graph composes \(N\) alternatives. Each of the alternatives contains \(M\) modules. Each module has several variants in its variety parameter (candidates). Within such a
hierarchical structure, we can easily identify multi-level configurations of subassemblies, intermediate parts, and component parts.

4.3.3 Operations on Alternatives: Finding the Optimal Solution

The major problem of Mass Customization Manufacturing is searching for the optimal configuration by which to satisfy the customer’s requirements and constraints. Constraints could be maximizing the benefit by finding the lowest cost alternatives, within the customer’s budget, or exclusiveness among modules due to size/function compatibilities. Finding optimal solutions in this configuration space is a NP-hard problem. Genetic algorithms (GA) (Holland, 1975) have now been widely adopted within such an optimal configuration search due to its ability of producing acceptable solutions involving a wide variety of configurations (Gen and Cheng, 2000; Jiao et al., 2007).

Genetic algorithms (GA) are computerized search and optimization methods that work very similar to the principles of natural evolution. Based on Darwin’s survival-of-the-fittest principles, GA’s intelligent search procedure is likely to find the globally best design or solution although it may converge towards a local optima in some problems. There is no general solution to avoid the local optima issue. The genetic algorithm maintains a
population of individuals, say \( P(t) \), for generation \( t \). Each individual represents a potential solution to the problem at hand. In the example of Figure 4.5, a \( P(t) \) is one instance of the variety parameter \( P \) vector (e.g., \( \{V_{1,1}, V_{1,2}, V_{2,1}, \ldots, V_{31,3-2}\} \)). Each individual is evaluated to give some measure of its fitness. Next, the GA generates and evaluates a new set of individuals. These new individuals are transformed from the old generation by means of genetic operations. Some are created by making change in the old individual, a process called mutation. Some are generated by combining parts from two old individuals, a process called crossover. This new set of individuals is then labeled the offspring \( C(t) \). Based on the parent generation of individuals and the offspring individuals, the GA forms a new generation of \( P(t+1) \). After several rounds, the algorithm converges to the best individual, which hopes to be the best optimal or suboptimal solution. A general structure of the genetic algorithm is as Figure 4.7 (Gen and Cheng, 2000).

Genetic algorithms encode the solutions to a problem into a genetic representation (called chromosomes). Based on his generic structure of the product family, Jiao et al. (2007) have proposed a general genetic algorithm (GGA), which uses a generic encoding scheme originating from the general structure with which to deal with diverse configuration spaces. The chromosome of the generic encoding is a finite-length string. Each fragment of the chromosome represents a module candidate. Each element of the string, called a gene, indicates a feature of the containing module. GGA is also deployed with a hybrid constraint-handling strategy to handle complex and distinct constraints at different stages of the evolutionary process. By adopting the customer-perceived benefit per-cost as the fitness function, CGA can solve product family design problems with regards to both customer preference and the manufacturer’s perspective (Jiao et al., 2007).
4.3.4 Summary of Alternative Representation in Product Family Design

A product family contains a set of product alternatives. The family can be represented into a tree hierarchy. The vector of parameters determines the configuration of the product. Products with different configurations form a configuration space. This configuration space has a clear boundary because the number of parameters and the set of values of parameters is predetermined.

The target of this design problem is to find a set of solutions satisfying the requirements and constraints. The set of solutions is a subset of all possible component assemblies in the space. From the perspective of a customer, the ultimate goal is to find the optimal configuration, which is the solution that satisfies the constraints, the requirements, as well as the optimality criteria (e.g., the cheapest). However, searching for the optimal requires the user to give a measurable optimistic criterion. In most cases, the user is unable to provide such a ruler. He/she may only be aware of a partial requirement (e.g., to buy a computer monitor, just know the budget limit and monitor size), or sometimes the requirement may be hard to measure (e.g., “good looking”). An optimal search that returns only a single result may not satisfy the user. It may be worth bringing in ideas from design space exploration. Based on the partially specified requirements, the system presents further choices to the user, such as possible optimal solutions for several configuration directions, and guides the user to clarify and provide descriptions for these missing specifications. At each step the user becomes aware of more choices and provides more specification. In the end(s) he is able to obtain a satisfying choice.

4.4 Applications and Representations

Design Galleries (Marks et al., 1997), subjunctive interfaces (Lunzer and Hornbæk, 2008), and Parallel Paths (Terry et al., 2004) propose three different concepts to allow users to generate, explore, and compare alternatives. In this section I examine each of them to see what kind of alternative representations can support these applications and how these applications may shape the representation.
4.4.1 Support Interaction Design in Parallel Paths

Terry et al. (2004) present Parallel Paths, an interaction model that facilitates generating, manipulating, and comparing alternative solutions. This model was demonstrated under the context of image manipulation by Parallel Pies, an interface that allows users to create, manipulate, and compare variations in a single working space. Here, alternatives are defined as solution variations, a user-designated set of distinct, alternative solutions to a given set of problems at a point in time.

In the interface of Parallel Pies, the center part is sliced into multiple sections, with each section showing a portion of one variation. The user can interactively adjust the slicing section to reposition and rotate to reveal different areas of the individual variations. Thus the main work area can selectively display all variations equally; emphasize one or more variations by giving them a bigger space; or simply display one variation as a whole (Terry et al., 2004).

In Parallel Paths, users interact with variations in the following ways (Terry, 2005):

- While executing a command, the user can choose the option of “Add Variation” to add the currently previewed result as a new alternative to the given solution. Previewing is also enhanced by Side Views (Terry and Mynatt), which can automatically generate sets of previews for each command parameter.

- The user chooses “create new variation” from a drop down menu. Any variation can be duplicated to create a new alternative.

- Each variation maintains a complete history of all its prior states, initially adopting the history of its source. Users can traverse the timeline to visit a prior state (called skating). Thus, users can duplicate a variation and return to a previous state to pursue an alternative path.

- Alternatives are embedded directly within the same solution workspace and are viewable through Parallel Pies.

- Commands are augmented to allow users to modify one or more alternatives simultaneously.
CHAPTER 4. REPRESENTATIONS OF ALTERNATIVES

Figure 4.8: Parallel Paths: Process diagram shows clear branching in Parallel Pie tasks

Most Related Representations Support Parallel Paths

Variations in Parallel Paths are very similar to the branches in Git. In Git, the user creates a new state of work with the command `commit`. In Parallel Paths, a new state is generated after the user issues a command. Both systems allow users to freely test out different solutions and ideas using variations (branches) during the process of developing an artifact (an image or software).

Along with Parallel Paths, Terry uses a Process Diagram (Figure 4.8) to visualize the user’s problem solving process (Terry, 2005). The left side of the tree represents states that the user visited. Edges show the evolution of variations. In this diagram, branches are very similar to Git without merging. The right two columns are called the active state timeline and command timeline. The active state timeline lists a set of active states at that point of time. The command timeline list all commands (such as adjust brightness, contrast, hue, etc.) issued by the user in temporal order.

The Git representation supports the following interactions in Parallel Paths:

- Create new variations. Similar to Git, these variations can grow from any past states in Parallel Paths.
- Skating. Users can traverse the timeline of a variation to retrieve any past state by following Git’s DAG or Parallel Paths’ tree diagram.

**Challenge of Adopting Git’s Representation**

In Parallel Paths, the user can choose to modify one variation or all variations in the workspace at once. This can be seen as the command timeline operating on one or more nodes in the state tree, which extends multiple branches simultaneously. However, in Git, if the user wants to embed a new feature into multiple branches, he/she has to create a new branch containing the new feature and then merge this branch with all other branches one by one. The user cannot modify the history inside one alternative timeline in Parallel Paths. As Terry et al. (2004) mentioned, history tracking tools such as Graphic Editable Histories (Kurlander and Feiner, 1990), would be able to nicely compliment Parallel Paths through the modification of previous commands with changes propagating down the timeline. At present, it is understood that neither the Git’s DAG nor the Parallel Path’s tree have the ability to propagate changes. As I have discussed before, the GIT structure does not allow for propagation because this process causes inconsistency to rise between objects and their SHA-1 names.

**Capabilities of Representations that Did Not Engage**

In Git, merge is one of the key operations that combines new ideas together. It would also be very useful in Parallel Paths. Merge causes branches to re-converge and to create a new object with all features derived from its parents. In image manipulation (or a similar design task), each variation has its own advantages. A “merged” version that combines proper variations may provide the user with a shortcut to achieve his/her design target. At the interface level, this merge could be done in the same workspace in Parallel Pies. It could be a simple mixture of several variations according to their weights (as are defined as slices in Parallel Pies). However, at the engineering level, we may be unable to discover a simple series of commands and parameters that can lead to the merged document. The parent documents may have very distinctive commands, and the parameters involved in commands may have no linear relationship with the result. Simple mixed commands and parameters derived from the parent documents may not be able to reproduce the merged document.
4.4.2 Support Interaction Design in Design Galleries

Design Galleries (Marks et al., 1997) aim to provide a solution for parameter tweaking problems in computer graphics applications. In many computer graphics applications, in order to generate a satisfying result, the user needs to find a set of proper values from a long list of parameters, and each parameter may have a wide range of choices. The combination of parameters may have a vast range of possibilities. Since generating a result based on one set of parameters can be expensive, the parameter-tweaking problem becomes practically intractable.

At first, through varying the given input parameters, Design Galleries automatically generate a large number of graphics/animations that cover the complete alternative space in a manner as broad as possible. This part requires a large amount of computation time, but can be done without a human’s presence. Next, Design Galleries select a small representative portion from the results based on perceptual differences. Finally, Design Galleries organize and arrange results in the interface based on similarity which will allow for easy browsing. Design Galleries have six key components: input vector, mapping, output vector, distance metric, dispersion, and arrangement. Input vector is a list of parameters (e.g., position of lights) that controls the generation of result graphics via a mapping process (e.g., ray trace rendering). The output vector is a list of values that summarizes the subjectively relevant qualities of the results. The distance matrix measures the perceptual similarity of outputs through the output vector. The dispersion function looks for a set of input vectors that map to a smaller but well-distributed set of outputs. Finally, the dispersed graphics are arranged based on the distance matrix, making these outputs perceptually grouped/clustered in the display.

Design Galleries provide two types of interfaces. One arranges output thumbnails in a hierarchical way. The other display arranges thumbnails in a 2D space that allows users to visually cluster them. The layout algorithm is based on a multidimensional scaling (MDS) method that maps high-dimension data onto 2D space.

Through these interfaces, users can select and compare alternatives in a well-organized browsing environment. The system automatically allocates and displays similar results once the user selects a result, which allows the user to approach a satisfying result. Then the system can easily retrieve the associated inputting vector of parameters, which can be used to produce the final production quality graphics or animation.
Most Related Representation to Support Design Galleries - GVS

The graphic alternatives in Design Galleries can be considered as a graphic family with a variety of input parameters. The generic variety structure (GVS) of Product Family Architecture (PFA) may be a good representation for Design Galleries. GVS uses a tree structure to represent the hierarchical structure of a product, components, and components with variety parameters. Leaves of the tree are potential values for each variety parameter, which can be used to construct the Design Galleries’ input vector. Combinations of the variety parameter decide the final product (the Design Galleries’ graphics). In GVS, different components are composed into the final product through the component hierarchy. In Design Galleries, parameters have to go through a mapping process to create the final image. We can consider GVS’s product component hierarchy equivalent to the mapping process of Design Galleries.

Challenge of Adopting the GVS Model

One important contribution of Design Galleries is the arrangement of output results for easy browsing. Both the hierarchical layout and the 2D mapping layout allow the user to perceptually distinguish/cluster the results, and thus this provides a visual cue for the user to drill down and find the best results. In order to adopt the GVS model, we may define the partition/hierarchy as a type of sub-space located inside of the big alternative space. Such a sub-alterative-space may be fitted into the configuration space (Figure 4.6), between the top level “configuration space” and the second level of individual alternatives. One alternative may belong to multiple subspaces, which will make subspaces overlap. Combined together, these subspaces compose an abundant alternative space.

Capabilities of Representations that Would Not Engage

Many problems have no quantifiable out value. The result can be inspected by a human, but it is impossible to be evaluated by the computer. This is the major reason why the inverse design (the computer searches for parameter settings to meet a user-supplied objective) does not work at this point. For this reason, those AI search (such as Genetic Evolution) algorithms are useless in this context.
CHAPTER 4. REPRESENTATIONS OF ALTERNATIVES

Other Useful Capabilities

GVS embeds constraints in its presentation. These constraints can be the incompatibilities of parameters among components, or constraints about domain knowledge. Such constraints may also be useful in the Design Galleries to reduce the number of possible input vector combinations. This can decrease the amount of computations and help users to access the result more efficiently.

The merge operation from Git could be useful in Design Galleries as well. The limited screen size is only capable of presenting a certain number of result thumbnails. This leads to a potential problem if the number of parameters is large or the range of each parameter is broad. There might exist millions of variations, but less than one percent (at most hundreds of variations) can be visualized in the gallery space. The difference between dispersed results will be large, which means that there may exist several next-to-best results but none of these fits the user's expectation. For this reason, the user may want to mix some of the selected candidates to obtain the best results. In Parallel Paths, each variation comes from a different series of commands. But in Design Galleries, all results are generated by the same mapping process. It is possible to calculate the proper values of the input parameters based on several selected candidates. A quick and easy rule might be: if the parameters are linearly related to the result, the input vector for the merged result could be the weighted average of input vectors of selected candidates. Such a solution can be seen as a hybrid approach of Design Galleries and interactive evolution.

4.4.3 Support Interaction Design in Subjunctive Interface

Subjunctive interfaces (Figure 4.10b) support users to setup, view, and control alternative scenarios simultaneously (Lunzer and Hornbæk, 2008). The motivation of subjunctive interfaces is to answer “what-if” questions, such as “What if I had asked the travel agent to check the alternative flight for the following day?” (Lunzer, 1998). In many applications, such as product purchasing, trip planning, or design creation, users want to see alternative scenarios and compare them. Initially, subjunctive interfaces aim to provide users with freedom to keep options open (users are free to specify querying parameters and combinations); ease of viewing a range of available outcomes; and ease of comparing outcomes and selecting preferred ones. Based on a series of studies (Lunzer, 1998, 1999; Lunzer and Hornbæk, 2003; Lunzer and Hornbaek; Lunzer and Hornbæk, 2008),
Figure 4.9: A subjunctive interface of querying census data (Lunzer and Hornbæk, 2008)
the current design principles for subjunctive interfaces are as follows:

- Set up multiple independent scenarios that exist at the same time. The user sets up multiple scenarios by selecting and combining input parameters. He has simultaneous access to these scenarios and is able to create, delete, or adjust scenarios.

- View scenarios side by side. The user is able to compare scenarios. Due to limited screen space, the number of scenarios presented together may be limited.

- Make changes to many scenarios in parallel. This principle improves the exploration efficiency and keeps alternative scenarios consistent.

Figure 4.10 shows two interfaces to let the user query census data. In a regular single mode interface (Figure 4.10a), the user can only see one set of results for a chosen combination of country, industry, and year. Within a subjunctive interface, the user is able to see several groups of result data together by setting up several combinations of menu items (Figure 4.10b). Through a series of mouse click (click, click+hold, mouse+key, etc.) interactions, the user can add, copy, delete scenarios, or update the inputting parameter of one or more scenarios. The user can also configure the layout of displays to compare scenarios.

Related Representations to Support Subjunctive Interface

Subjunctive interface provides menus for the user to select and set up scenarios. I consider these menus as parameters, and the items in one menu are possible values of the parameter. The combinations of menu items construct the space of alternative scenarios. It is possible to let the computer automatically generate all possible combinations and display them. Sometimes the user only desires to focus on several variants of one parameter.

Figure 4.10 shows the flow from parameters to outputs. This flow also fits Design Galleries. In subjunctive interface, the construction of parameter vectors are done through the user’s manual interaction, while in the Design Galleries it is through automatic generation. The input parameters control the output though a mapping or query process. Since Design Galleries automatically generate a large number of combinations, it has to use the disperse function to reduce the number of outputs for display. Users of Design Galleries are more interested in finding the best result (in order to find an appropriate set of input parameters). However, in a subjunctive interface, users are more interested in
manually selecting inputs to compare the outcomes. Similar to Design Galleries, a simplified generic variety structure from the product family design may also be used here. But the AI search algorithms are not needed in these subjunctive interfaces.

One principle of subjunctive interfaces is that of making changes to many scenarios in parallel. This means that the user can update all values in one or more columns of the parameter vectors at once. A similar operation also exists in Parallel Paths, in which one command can be applied to one or more variations simultaneously. The Parallel Paths’ representation model might be informative for a subjunctive interface design.

**Partial Scenario in Subjunctive Interface**

In order to setup a scenario, it seems that the user has to select items from each of the menus. According to the literature review (Lunzer, 1998, 1999; Lunzer and Hornbæk, 2003; Lunzer and Hornbaek; Lunzer and Hornbaek, 2008), there is no evidence showing that a user can set up a scenario by only specifying a portion of all parameters. However, the user may wish to have such an ability. By using the census data example, the user may desire to examine the data annually for all countries and industries in whole, or examine data on an annual basis for each individual country and industry. The user does not need to define the full scenario by selecting all involved parameters at once. At first, he can setup the scenario with one parameter (such as the year), and then he can gradually add other parameters. The “partial” scenario may provide various suggestions to the user so that he may add more relative parameters in order to make this scenario more detailed and complete.
4.5 Conclusion

At this point I have discussed alternative-support mechanisms in six applications: Git (software configuration management), design space exploration, product family design, Parallel Paths (image manipulation), Design Galleries (tweaking parameters for computer graphics), and subjunctive interface (explore different scenarios). These applications share many common features, but they also have their distinctive characteristics.

Many digital applications provide the possibility for users to set up, choose, and modify alternatives. Because of the variety that exists both within the tasks and application context, the ways that users will define alternatives and their goals to use these alternatives will differ. For example, some users will desire to find an optimal solution for a requirement, while others may wish to see and compare alternatives purely based on their own judgement.

In the above sections, I have discussed the representations of alternatives in three different domains.

- The software configuration management system Git adopts a directed acyclic graph to capture and represent branches and history states. The DAG faithfully represents the process of software development.

- Design space exploration utilizes typed feature structure’s ability for handling partial information to represent the complex design space in a computable and formal way. Designers are able to explore design alternatives, starting from a partially described design and then adding details step by step.

- The general variant structure of product family design uses a AND/XOR tree to represent the components hierarchy of a product, variety parameters, and constraints among parameters. With a given specification, the genetic algorithm is able to find the near-to-optimal combination of configuration parameters.

We can see that these three representations distinctively differ from each other. Although these representations may complement each other, as of yet there does not exist a universal representation that can serve as a general device to handle most of the alternative operations.
Chapter 5

Design Science Research as the Methodology

In general, my research aims at methods to handle alternatives in computer systems. Any formalism has to be evaluated to ensure it is at first mathematically correct on concepts and algorithms and then useful for real cases. These evaluations can then lead me to reflect back to improve the formalism design, which makes the design-evaluation an iterative process.

This research is inspired by both computer-aided design and visual analytics, especially the CZSaw system, which aims to analyze entities and relations. Both parametric CAD system and CZSaw use propagation-based models as the primary representation. My research is built based on such propagation-based parametric models.

5.1 Possible Research Methodologies

In 1992, Galliers (1992) defined the following researching methodologies in information system research:

- *Laboratory Experiments*: Control and study a small number of variables intensively
- *Field Experiments*: Realism at the expense of control
- *Surveys*: Time-stamped samples from which inferences are made
- *Case Studies*: Descriptive reports of projects/episodes
• **Grounded Theory**: Look for patterns in collected situational data

• **Action Research**: Researcher participates directly in a project

• **Subjective/Argumentative Research**: Idea generation

Mingers (2001) argued that research results will be richer and more reliable if different research methods, preferably from different (existing) paradigms, are routinely combined together. The ways to mix methods include:

• Methods employed in sequence

• Methods carried out in parallel with results feeding into each other

• One as the main approach with contributions from others

• Combinations of methods developed specifically for the task

• Research conducted simultaneously at different levels of an organization and using different methods

Laboratory experiments, action research, and the subjective/argumentative method relate to my research. Laboratory experiments can identify precise relationships between chosen variables via a designed lab situation, using quantitative analytical techniques, with a view to make generalizable statements. In this research process, designed algorithms may be evaluated by using laboratory experiments to check their correctness.

The Subjective/Argumentative method focuses on idea generation. It is a creative research method based primarily on opinion and speculation. It is useful in building a theory that can subsequently be tested. The researcher plays a much more important role in this method. However, this method has an unstructured, subjective nature. Due to the highly emphasized role/perspective of the researcher, it has a likelihood to introduce biased interpretations. Since the key point of my research is to create a formalism, such idea generation is very important.

The action research method allows the researcher to participate directly in a project. It gives the study group practical value, adds to theoretical knowledge, and enhances the competence of the actors. But this method has risks, such as: lack of control over variables, restriction to single projects and organizations, openness of interpretation, biases/omissions in description, and competition of objectives. Action research is normally
completed in cycles. Each cycle starts by engaging with a real world setting/situation. Then the researcher defines the issue (initial observations and existing data), plans actions/interventions, takes actions/interventions, and finally analyzes and reflects on previously taken actions/interventions. Then, the next cycle starts by re-engaging with real world settings/situations once again. After each cycle, the researcher produces reports of findings.

5.1.1 Two Research Paradigms

Walls et al. (1992) and Hevner et al. (2004) distinguished two research paradigms in the information systems discipline: Behavioral Science and Design Science. In the behavioral science paradigm, researchers seek to develop and verify theories that explain or predict human or organizational behavior. In the design science paradigm, researchers seek to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts.

Traditionally and very generally, research is an activity that contributes to the understanding of a phenomenon. Design is the work of invention and bringing into being. Some examples of artifacts in information systems include (but are not limited to) algorithms, human/computer interfaces, system design methodologies, and languages. In recent years, design activities have been brought into focus at an intellectual level. Simon (1996) makes a clear distinction between "natural science" and "science of the artificial" (also known as design science). In natural science, a body of knowledge is about some class of things, like objects or phenomenon, in the natural or societal world. The knowledge describes and explains how these things behave and interact with each other. In artificial science, the body of knowledge is about man-made artificial objects that are designed to meet certain desired goals.

Design science research is fundamentally a problem solving paradigm, with emphases on problem solving and performance improvement. Takeda et al. (1990) illustrates a general design cycle (Figure 5.1) in which each design begins with an awareness of a problem. The design process steps include an awareness of a problem, suggestion(s), development, evaluation(s), and conclusion(s). Development, evaluations, and further suggestions are frequently iteratively performed over the course of the research. During any phase of development, evaluation, or conclusion, the researcher may encounter/understand more
about the problem, which will loop the cycle back to start another round of design.

In information systems, as a problem solving paradigm, design science research has its roots in engineering and the artificial sciences. It involves the analysis of the use and performance of designed artifacts to understand, explain, and, very frequently, improve the behavior of aspects of information systems (Vaishnavi and Kuechler, 2004).

5.2 Design Science Research

5.2.1 The Research Question

A research question is the question that the research study sets out to answer. Differing types of research questions lead to different research methods. Table 5.1 lists the main types of research questions from Meltzoff (1998).

We know that a parametric model can enable people to reuse it by assigning different values to the independent properties (parameters of the model). But, currently there is no easy way to handle a large set of parameters. Therefore, I choose my research question as a design question: What is an effective way to help people construct, understand, and explore alternatives for a parametric model? Following this question is the evaluation question: Does the designed artifact (representation and algorithm) satisfy the goal?
Type | Questions | Answers
--- | --- | ---
Exploratory | Does X exist? What is X like? How does X differ from Y? | A clearer understanding of the phenomena, including more precise definitions of the theoretical terms, evidence that we can measure them, and evidence that the measures are valid
Base-rate | How often does X occur? How does X normally work? | Normal patterns of occurrence of the phenomena
Relationship | Are X and Y related? How are X and Y related? | If and how to phenomena are related
Design questions | What is an effective way to achieve X? Which strategies help achieve X? | Describes better ways to solve some problems or situations

Table 5.1: Main types of research questions. Adapted from (Meltzoff, 1998)

5.2.2 Design Science Research as My Research Methodology

Following Takeda et al. (1990)’s design cycle, we specify a methodology of design research by defining the outputs from each step. At the phase of awareness of problem and suggestion, the output is a proposal for a new research effort. The suggestion phase produces a tentative design that is intimately connected with the proposal. This step is a creative step and has been criticized as introducing non-repeatability into the design research method. Next, in the development phase, the tentative design is implemented. Once constructed, the artifact is evaluated according to the criteria set by the proposal. The evaluation results, insights gained in construction, and use of the artifact are brought together and fed back to another round of suggestions. Finally, a conclusion is drawn; the design effort is consolidated and written up. More important, the knowledge gained from the research is confirmed so that it can be repeatedly applied, or anomalous behavior found in the process that can serve as the subject of further research.

According to the five-step research cycle (Figure 5.1), the procedure of my research is:

- Awareness of a problem. Understanding the needs of handling alternatives and knowing the current status of other research. This starts with personal experience and is followed by a literature review.

- Suggestion / Tentative design. We need a formalism as the representation of alternatives. This is the creative part of the research. Our promise is that operating on
variations spaces may give the user tools to manage alternatives.

- Development. I use mathematical statements and specifications to formally describe the mathematical model and algorithms.

- Evaluation. At first, the formalism has to be mathematically solid and clear to make the design valid. Second, it has to be useful in real cases. Feedback information is collected during the evaluation process and taken back to improve the formalism design.

- Conclusion. Where I summarize and question the knowledge gained from this research.

5.3 Making the Research Solid

To make the design activity valid as design science research, Hevner et al. (2004) raises seven guidelines:

- Design as an artifact: The purpose is to produce a viable artifact in the form of a construct, a model (abstractions and representations), a method (algorithms and practices), or an instantiation (implemented and prototyped systems).

- Problem relevance: The objective is to develop technology-based solutions for important and relevant business problems.

- Design evaluation: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.

- Research contributions: Effective design-science research must provide clear and verifiable contributions in the areas of the design artifacts, design foundations, and/or design methodologies.

- Research rigor: Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.

- Design as a search process: Design science is inherently iterative. The search for the best, or optimal, design is often intractable for realistic information systems problems.
Communication of research: The research work must be published in both the academic community and in the practitioner’s community.

March and Smith (1994) emphasized evaluation as one of the two activities in design science: build and evaluate. Hevner also treated evaluation as "crucial". As my research tries to find a new way to handle alternatives, it is a design question in the form of either: "What’s an effective way to achieve X?" or "What strategies help to achieve X?". The outcome will be a formalism as the representation of alternative in the area of visual analytics and computational design. The formalism comprises definitions of objects and algorithm of handling these objects. The evaluation question in my research determines whether the formalism is able to represent and handle alternatives. It is not about measurements, such as are evaluated by "how much performance improves" or "how the user likes it" questions.

5.3.1 Design Evaluation Methods

According to the environmental settings, Venable (2006) distinguishes evaluation can as either an Artificial Evaluation or a Naturalistic Evaluation. Truly the best evaluation should be done by Naturalistic evaluation with real users using real systems to solve real problems. However, due to resource limitations, frequently the evaluation has to be artificial, where one evaluates the solution in a contrived and non-realistic way, which is the most likely method for my research.

Hevner et al. (2004) categorized design evaluation methods as follows:

- Observational
  - Case Study: Study the artifact in depth in the business environment
  - Field Study: Monitor use of the artifact in multiple projects

- Analytical
  - Static Analysis: Examine the structure of the artifact for static qualities (e.g., complexity)
  - Architecture Analysis: Study the fit of the artifact into technical architecture
  - Optimization: Demonstrate inherent optimal properties of the artifact or provide optimality bounds on artifact behavior
- Dynamic Analysis: Study the artifact in use for dynamic qualities (e.g., performance)

• Experimental
  - Controlled Experiment: Study the artifact in a controlled environment for qualities (e.g., usability)
  - Simulation: Execute the artifact with artificial data

• Testing
  - Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects
  - Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation

• Descriptive
  - Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact’s utility
  - Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility

Among these evaluation methods, analytical methods and simulation are artificial evaluation methods. All other methods can be done both ways in either artificial evaluation or naturalistic evaluation.

Evaluating My Research

The evaluation of my research is to answer the question of whether the formalism is able to represent and handle alternatives.

At first I will test the formalism by simulation. The artificial data can be designed to test out all perspectives (e.g., algorithms) of the formalism. Next I will use the descriptive scenario method help people to understand the formalism by means of a thick, detailed description of the usage and scenario of the formalism.

It would be ideal to involve outsiders in order to analyze, evaluate, and reflect upon the formalism during or after the development process. A formalism is not a working
system, but rather the underlying machine enabling the system. The formalism has to be implemented inside a system for the user to test. Due to limited resources, it is very hard (if not impossible) to test this formalism by having real users working with real data in a well developed system. Also, it will be hard to design the evaluation. There are many confounding variables such as the design of the system’s interface (layout etc.) and interactions within the interface. A poorly designed interface may ruin the users’ experience and may lead to bias toward the evaluation of the formalism.

The IEEE VAST (Visual Analytics Science and Technology) challenge provides many close-to-real data and tasks for the researchers to use. I use some of the VAST tasks to construct detailed scenarios around the artifact in the second simulation in order to better gain insight into the system over a potential real life environment.

5.3.2 Ensure Research Rigor

The rigor of research work is determined by its validity and reliability. Creswell (2003) lists several strategies to ensure rigor, such as: triangulation, member checking, thick descriptions, clarification bias, reporting of discrepant information, prolonged contact with participants, peer debriefing, and external auditing. I will primarily rely on data triangulation. Among the five major types of triangulation (data, investigator, theory, methodological, and environmental), data triangulation and methodological triangulation will be used in this research.

The simulation method will be used to answer the question about whether the formalism is logically correct. In this stage, investigator triangulation will be used to ensure rigor. I will invite another researcher to conduct peer reviews on the formalism and the simulation system.

The descriptive scenario method will be used to answer the question about whether the formalism is useful for handling alternatives. I will study the following three data sets with CZSaw:

- 2008 VAST cellphone mini challenge (categorical data in spreadsheet table)
- 2009 VAST social network mini challenge (social network data in spreadsheet table)
- 2010 VAST firearms mini challenge (text data with entities extracted by external applications)
Working on these data sets has aspects of minor action research. The researcher (myself) actively participates in the project. While analyzing the data, I will have to study and grow to understand all data sets, which leads me to a better awareness of the problem. Also, the lessons learnt from one data set may be used to improve the design, which will lead to better results in the next data set.

5.3.3 Developing Prototypes for Evaluation

To expand, confirm, and understand the formalism, I implement two prototypes in two stages. One stage involves developing a simulation system and using the simulation system to test, understand, and enrich the formalism. The second stage is my work with CZSaw to demonstrate its handling of alternatives in visual analytics systems with the chosen IEEE VAST data sets.

Nielsen (1994) places prototypes into two types: horizontal and vertical. A horizontal prototype is generally a user interface prototype. It provides a broad view of the system, focusing more on user interface and interaction rather than on functionalities. The vertical prototype focuses on a single subsystem or function; it is useful to obtain detailed information. Clearly for my purposes, both prototypes are vertical prototypes that deeply focus on handling alternatives to explore and confirm the designed formalism.

There are several well-established prototypical methods, one being the Dynamic Systems Development Method (DSDM) (Consortium, 2011). The DSDM is an ISO 9001 approved agile project delivery framework. Its iterative and incremental approach encourages continuous user/customer involvement. The DSDM categorizes prototypes into four categories (Consortium, 2011):

- Business prototypes – used to design and demonstrate the features or functionality being developed.

- Usability prototypes – used to define, refine, and demonstrate user interface design usability, accessibility, look and feel, and such aspects as security, and availability (non-functional requirements).

- Performance and capacity prototypes – used to define, demonstrate, predict, and test how systems will perform under peak loads as well as to demonstrate and evaluate, for example, volume-handling, response times, and load bearing.
• Capability/technique prototypes - used to develop, demonstrate, and evaluate a design approach or concept. Often called a proof of concept prototype.

My first prototype is to simulate the formalism, which demonstrate the features formalism including basic concepts, objects, and operations. It is a business prototype. My second prototype is to demonstrate and evaluate the usage of the formalism on research data sets, which is a capability/technique prototypes. To better understand the characteristic of the formalism, in the first prototype, I will use 3-D geometric data as the artificial data to test diverse perspectives of the formalism. By visualizing the results in a 3-D screen, we can visually inspect and understand the input data and results generated. For the second CZSaw part, the main purpose is to demonstrate how to utilize this formalism to enable uses handling alternatives. Due to the nature of the data and the task, only a portion of the formalism is needed for these visual analytics tasks.

5.4 Summary

After comparing several research methodologies from information systems research, I have adopted design science research as the main methodology. Design science research is fundamentally a problem solving paradigm, with emphases on problem solving and performance improvement. The design process is an iterative circle that consists of the awareness of a problem, suggestion(s), development, evaluation(s), and a conclusion.

Evaluation is crucial in the process. I deploy two evaluation phases for the designed formalism. The first phase is a simulation prototype to demonstrate and study designed concepts and algorithms by using artificial data. The second phase aims to understand utility of alternatives in somewhat realistic environments. These two phases of evaluation are methodology triangulation. The first one also uses investigator triangulation to ensure the rigor of the simulation system. In the second phase, I employ this formalism on three different tasks and data sets, which is data triangulation.
Chapter 6

Design of the Formalism

Through the previous literature review, I have outlined the need for handling alternatives and the current status of other research. After working with CZSaw and observing the potential and problems of reusing the CZSaw symbolic model, I refined the research question as: “What is an effective way to help people construct, understand, and explore alternatives for parametric models?” To answer this question, I have chosen to design a formalism as the representation of alternatives in parametric models. It is hard to represent alternatives in general. To make this research more focused, I further refine my target formalism as a representation for variations in parametric models.

6.1 Variations

6.1.1 Alternatives and Variations

Alternatives are multiple solutions for a given problem or design with different parameters. They are multiples states of the model. Atman et al. (1999) and Akin (2001) have shown that expert designers develop more design alternatives than novices during their design process. Similarly, in visual analytics, more alternative solutions for one task may result in better solutions.

More precisely, we shall distinguish between the concepts of alternative and variation.

For a given problem or design, we define different solutions (e.g., different procedures or algorithms) of the problem as alternatives. The same procedure or solution, it may be
applied with different parameters. We have defined these similar solutions with different parameters as variations. In the context of our symbolic model, the graph structure (nodes as objects or interim results, edges as connection among nodes, etc.) represents the solution of the problem.

A parametric model is a labeled directed graph. We label the graph-independent properties (Woodbury, 2010) of a graph as its variation properties (values of node A, B, and C in Figure 6.1). The variation values of a graph are the values of these properties. By definition, these properties are contained within source nodes by definition.

Constraint expressions determine the label and direction of edges. We can illustrate alternatives and variations in this research with the following examples:

Two non-isomorphic labeled graphs are alternatives if they have identically named source nodes, with identical variation properties within each node.

Two isomorphic graphs are variations if they differ only in the values of their variation properties. Such two graphs match one-to-one on nodes and edges, which means that the design solutions represented by these graphs are exactly the same.

Nodes A, B, and C are all single property (value) nodes. Values of A, B, and C (graph-independent properties) are variation properties. A subset of {A, B, C} is a variation head of the graph.

**Figure 6.1:** Variation properties and variation head of a graph

### 6.1.2 Variation Head

Variation properties represent inputs to a graph. We are interested in exploring different scenarios that differ in their inputs. Thus, we further concentrate our focus onto a subset of the graph nodes. We call such a subset of nodes a variation head of the graph, which represents some of inputs to the parametric model.

A variation head of a graph is a subset of its source nodes, with each node containing
a subset of the variation properties of the respective source node (Figure 6.1). A variation head is an unconnected graph. Since variation properties do not hold any constraint expressions as their values, these nodes will not connect to each other. We deal with variation heads because they represent the inputs for a graph, and we are interested in exploring scenarios that differ only in their inputs.

An alternative body of a graph is the graph with some of its variation properties unbound. The minimal alternative body is the graph with all of its variation properties unbound.

Two graphs are variation compatible with respect to a graph $G$, if $G$ is a variation head for both graphs (Graph a, b, and c in Figure 6.2).

Graph a, b, and c are variation compatible graphs regarding variation head \{A,B,C\}

In a strict variation head each variation property contains its respective variation value. This implies that a variation head can hold property values that differ from the graph from which it was originally derived; we can change a property value in a variation head, and it remains a variation head for the graph from whence it came. All variation heads have no edges; as they are unconnected, any graph lacking in edges can potentially be a variation head for another graph.

The maximal variation head of a graph is the variation head comprising all source nodes and all variation properties for each. One special case of a variation head is that all of its properties are unbound, which means that, with respect to a variation compatible graph, we know the variation structure but not its values. Sometimes we may only know one or several properties of a variation head, which means that we only know part of the graph structure. For some heads we only know one property, an call such a head single property variation head. Knowing a variation property but not its value (unbound) is different than not knowing the variation property at all.

We use the dot notion “.” to note the variation head, nodes in the head, and properties
within a node. For example, \textit{VariationHead.NodeName.PropertyName} refers to the property \textit{PropertyName} of the node \textit{NodeName} in the variation head \textit{VariationHead}.

We note that a variation head is a collection of nodes like \{node1, node2\}, which can be expanded to \{node1:[Property = Value],node2:[Property = Value],\ldots\}. For example, \{A : \{p = 1, q = 2\}, B : \{r = 3\}\}. If we use the dot path notation, the example variation head can be written as \{[A.p=1, A.q=2],[B.r=3]\}.

I use attribute-value matrix (AVM) notation to diagrammatically represent variation heads. Slots are the variation properties and their values are written next to the properties (Figure 6.3).

\[
\begin{align*}
A : & \begin{bmatrix} p : 1 \\ q : 2 \end{bmatrix} \\
B : & \begin{bmatrix} r : 3 \\ s : 4 \\ t : 5 \end{bmatrix}
\end{align*}
\]

\textbf{Figure 6.3:} The AVM notation of a variation head with two nodes

A state \textit{G} of a parametric model is the graph with values assigned to properties. We define the following function over any state \textit{G}:

\[
\text{propertyV}(G, p)
\]

returns the variation head of \textit{G} in which the variation property \textit{p} contains the value of that property from \textit{G}, with all other variation properties unbound.

In the following sections, for convenience, when the node name is not important in the operation, I will use a simple form by omitting the node name and only keeping the property name, such as \[ p : 1 \], \[ p : 2 \\ q : 3 \] to denote variation heads. In such cases, property \textit{p} and \textit{q} could belong to either the same node or different nodes.

\subsection{Variation Spaces}

A variation space is an ordered collection of variation heads. There is clearly a very large number (infinite if any property has an infinite domain) of possible variation heads for a given graph. The powerset of these heads ordered in any way gives the possible variation spaces for a graph. To define and explore such spaces we need to generate variations, filter variations, and navigate from one variation to another.
Generating variation heads can be accomplished in two ways: one is to directly assign values to a variation head of a graph, while the other is to combine two variation heads to create a new variation head. For such combinations, we use the unification operator (see Section 6.2 below).

Once we have a variation head, we can combine it with the alternative body of a graph, which will drive the graph to a certain state. Combining variation heads in a variation space with the alternative body, we get a space of states.

### 6.2 Operations Over Variation Heads

#### 6.2.1 Unification

Unification (⊔) is the process of combining information. It produces a new object that carries information both (and only) from its input objects. Here we are concerned with combining variation heads as those parts of a graph which differ only in their independent variables. Variation heads are unconnected by definition, so their combination does not involve combining constraint expressions (the source of graph edges). Unification will also allows us to combine variation heads with alternative bodies.

By convention we use $A \ldots D$ to designate nodes and $G \ldots J$ to designate graphs.

Here are some unification basics:

Unification, if it exists, is unique.

Typically, we define unification over a set of objects. By convention, each such set has a single most general element, called bottom and denoted as ⊥ (Carpenter, 1992, p.12). ⊥ serves as a zero element. If a set lacks such an element, we simply add it.

$$A \sqcup \perp = A$$

Other unification properties:

$$A \sqcup A = A$$

Unification may fail. Again, by convention, we introduce an element called top (denoted as ⊤) to represent the result of such failure. If two objects $A$ and $B$ are not unifiable, we write

$$A \sqcup B = \top$$

Of course,

$$A \sqcup \top = \top$$
In a set $S$ supporting unification, the following principles are true.

\begin{align*}
\bot &\in S \\
\top &\in S \\
A, B &\in S \text{ and } A \sqcup B = C \implies C \in S \\
\end{align*}

Unification is commutative.

\[ A \sqcup B = B \sqcup A \]

Unification is associative.

\[ (A \sqcup B) \sqcup C = A \sqcup (B \sqcup C) \]

Consider two nodes, $A$ and $B$. $A \sqcup B$ is another node $C$, which has all of the properties in $A$ and $B$, with the values of each property unified. For example, $A.s \sqcup B.s = C.s$. Properties in $A$ and not in $B$ (and vice versa) are taken directly into $C$.

\[
\text{name}(A)
\]

returns the name of node $A$.

Consider two variation heads $G$ and $H$, that is, two unconnected graphs. Remember that nodes in a variation head have unique names. We define unification over $G$ and $H$, with $G \sqcup H$ as the variation head $I$ such that:

- For each node $A$ in $G$, with name$(A) = p$ and $B \in H, \text{name}(B) = p$, $I.p = G.p \sqcup H.p$.
- For all other nodes $A \in G$, with name$(A) = p$, $I.p = G.p$.
- For all other nodes in $B \in H$, with name$(A) = p$, $I.p = H.p$.
- In above, $G.p \sqcup H.p$ is the unification of two nodes.

In the AVM notation, the unification result of two variation heads will be written as:

\[
\left[ \begin{array}{c} p:1 \\ q:2 \end{array} \right] \sqcup \left[ \begin{array}{c} p:1 \\ q:2 \end{array} \right] = \left[ \begin{array}{c} p:1 \\ q:2 \end{array} \right].
\]

### 6.2.2 Subsumption and Unification of Order Types

Subsumption measures information. For two objects, $F_1$ subsumes ($\sqsubseteq$) $F_2$ if and only if all the information contained in $F_1$ is also contained in $F_2$. The notion of subsumption, or information containment, can be used to define the notion of unification, or information combination. If $F_1 \sqsubseteq F_2$, then $F_1 \sqcup F_2 \Rightarrow F_2$.

Order types in Chang (1999)’s Ph.D dissertation define the subsumption relation of explicit values. In Chang’s definition, order types are partially ordered data sets that satisfy the subsumption relationship. His order types are defined under the context of typed
feature structures (Carpenter, 1992, p.13). We borrow this concept to simply measure the information carried by the value of graph-independent properties. The most general element contains the most general information, which is the bottom ($\bot$). Some of the order types mentioned in his work include the following:

- The lifted numbers: $a \sqsubseteq b$ if and only if $a = b$ or $a$ is $\bot$
- The ascending numbers: $a \sqsubseteq b$ if and only if $a \leq b$ or $a$ is $\bot$
- The descending numbers: $a \sqsubseteq b$ if and only if $a \geq b$ or $a$ is $\bot$
- The intersection intervals: An interval is a collection of numbers with lower and upper bound. It is written as $[a_l, a_u]$ that $a_l$ and $a_u$ denote as the lower and upper bound. $a \sqsubseteq b$ if and only if all members in $a$ are also in $b$.

With order types, unification results become semantically rich. For example, to compare a product’s price, at least $\$10 \sqsubseteq \$20$. Here, these values ($\$10$ and $\$20$) are ascending numbers. Then the unification of above information is the price is at least $\$20$.

In another example, let $A_1$ and $A_2$ be two integer intersection intervals. $A_1 = [2, 100]$ that means $2 < A_1 < 100$, and $A_2 = [0, 5]$ that means $0 < A_2 < 5$. Then the unified result of $A = A_1 \sqcup A_2 \Rightarrow 5 > A > 2$, which is, $A = [2, 5]$.

In the context of intelligent document analysis, location (address, city, province/state, country), time (second, minute, hour, day, month), human (firstname, lastname, name) could all be order types. For example, May 2011 $\sqsubseteq$ May 1st, 2011 $\sqsubseteq$ 11:15am, May 1st, 2011, address ”BC, Canada” $\sqsubseteq$ ”Vancouver, Canada”.

To start our discussion simply but without losing generality, in the following demonstrations, I use integers as values of the properties unless indicated. For two integers, $i_1 \sqsubseteq i_2$ if and only if $i_1 = i_2$ or $i_1$ is $\bot$. Two different integers unify to top ($p \sqcup q = \top$ if $q \neq q$).

### 6.3 Notation of the Variation Space

#### 6.3.1 A Simple Example

A variation space for a graph $G$ is an ordered collection of some of the possible variation heads of $G$. Each head can be specified as a set of named nodes, with property values
defined. However, such a notation would be awkward to use; it would involve repeatedly specifying objects and their properties. It is useful to have a more compact notation. The replication concept from GenerativeComponents™ (GC) provides us with a beginning example.

In the symbolic model, the graph independent property can be assigned by a list of values (an array). It is called a replication in GC’s implementation. GC automatically evaluates each of the values in the replication list, and it populates all successor nodes’ dependent properties. The dependent properties’ evaluated value will be the resulting replication list. Each individual element in the independent properties has a corresponding element in its successor’s dependent property. If the dependent property has more than one predecessor independent properties holding replications, then GC provides two options: AllCombinations and CorrespondingIndexing to generate input tuples. Using an example, nodes A and B are both single property (value) nodes with integer values, where \( A = \{1, 2, 3\} \) and \( B = \{100, 200\} \). Node C is a 2D point with coordination \( x \) and \( y \). \( C.x = A; C.y = B \). For convenience, I simply use the node name such as \( A \) to refer to its value instead of the full notation with value property like \( A.value \). The result of two different options is illustrated below:

- **AllCombinations.** With this option, the evaluated values of C are 6 points of (in the form of \( \{x, y\}\)) \( \{\{1, 100\}, \{1, 200\}, \{2, 100\}, \{2, 200\}, \{3, 100\}, \{3, 200\}\} \). This option will create all possible combinations by iterating elements in each replication list.

- **CorrespondingIndex.** With this option, the evaluated value of C are 3 points of \( \{\{1, 100\}, \{2, 200\}, \{3, 200\}\} \). This fetches elements from every replication list by the same index. The result will have the same length as the minimum length replication list. GC ignores non-corresponding elements from the longer list.

### 6.3.2 Initiate the Space

GC’s replication ability over independent properties provides us with a start from which to build variation spaces.

In analogy with replication, the graph with

\[ \{A.p = \bullet 1, 3, 6 \bullet\} \]

denotes the set of variation heads

\[ \{\{A.p = 1\}, \{A.p = 3\}, \{A.p = 6\}\} \]
This set of variation heads can be seen as a variation space, although it is simple, that has only the property $p$ of node $A$. A graph may have more than one independent property. Properties’ values can be either defined by the user explicitly, or values can be gathered based on the description of the given problem.

We build the variation space starting from spaces with single property variation heads. Each space contains variation heads for the same graph independent property. For a graph with multiple ($N$) graph independent properties, we will have $N$ initial single property variation spaces.

Next we define several operations over variation spaces. Variation spaces that contain more complex variation heads can be built from these simple spaces.

### 6.3.3 Structure of a Space

In a variation space, the order of elements (variation heads) and the positions of the elements in the space are important. The user may think some values are more important and intentionally put them in front. For single property variation spaces, these variation heads are created by the user (or the system) by assigning a list of values to one independent property and leaving others unbound. The order of values in the initial list may be meaningful, which makes the order in the variation space meaningful. For two different variation spaces, the index must be co-related in order to have a one-to-one match. When we talk about operations on variation spaces, by default, we keep the order information. In one space, there may exist multiple variation heads with the same property values although located in different positions.

We use a numerically indexed array (key/value pair structure) to organize variation heads in a space, and we adopt array notation for variation spaces. The simplest ordered space is a list of variation spaces in a linear sequential order (a one dimensional array), e.g., $\{A.p = 1\}, \{A.p = 3\}, \{A.p = 6\}$, and by default, we assign a sequence of integers, starting with 0 as the key for these elements. Of course, all index sets can be mapped to a set of integers. We call such a space a linear space or 1-D space, denoted as a list of variation heads inside $\{ \ldots \}$ or a one row table in a figure (Figure 6.5a). Normally a single property variation space is $1-D$ space.

Operation on such an indexed array structure is efficient. In term of time complexity, the operation of search, insert, and delete are all $O(1)$. The storage space is $O(n)$. 
We use $V[i]$ to denote a variation head in the space $V$ at the index of $i$. For a given space $V$, we can directly assign elements into certain positions, as shown in figure 6.4.

$$V[0] = \{A.p = 1\}$$
$$V[2] = \{A.p = 3\}$$
$$V[5] = \{A.p = 8\}$$

**Figure 6.4:** Assigning variation heads into the space $V$

An ordered space may have a higher order just like a multi-dimensional array. The index is a tuple of non-negative integers. We name such a higher-order space as an $n$-dimensional space or an $n$-D space. The index in an $n-D$ space is a tuple of integers, denoted as $[i_0, i_1, \ldots, i_n]$. $V[i_0, i_1, \ldots, i_n]$, which can be used to retrieve the variation head in the space $V$ at that index. Similar to the array definition, the index of the first element in the dimension is 0. If including the index, a space can be written as \{[index_0]: { \ldots },{index_n]:{ \ldots }}\}. For example, a 2-D space can be written as \{[0,0]: {A.p = a, A.q = 1}, [0,1]: {A.p = a, A.q = 2}, [1,0]: {A.p = b, A.q = 1}, [1,1]: {A.p = b, A.q = 2}\}. For convenience, we can also denote a $n$-D space as nested curly baskets \{\{\ldots\},\ldots,\{\ldots\}\}, with elements in higher dimensions nested in lower dimensions. Above 2-D space example can be write as \{\{\{A.p = a, A.q = 1\}, \{A.p = a, A.q = 2\}\}, \{\{A.p = b, A.q = 1\}, \{A.p = b, A.q = 2\}\}\}.

Below I use a grid table to demonstrate variation space and operations (Figure 6.5). In the table representation of a 2-D space, according to an element’s location, its row number is the first dimension, and the column number is the second dimension. A higher dimensional array can be illustrated as nested tables. Figure 6.5(c) shows a 3-D space.

### 6.3.4 Sparse Space

In computer science, a **Sparse Array** is an array that has contents that are lower than its maximum size or in which the array has free or empty locations. The similar case may happen to our variation space. The space may have some positions (indices) empty. If we need to access such an empty position, the result is *null*, which means that there is no object in there. *Null* does not mean $\bot$. $\bot$ means there is an object, but we do not have any information about it. In our table figure representation, we will use empty cells to mark these positions (Figure 6.6).
CHAPTER 6. DESIGN OF THE FORMALISM

6.3.5 Index Set

By default, our space is an array, and elements in the space are ordered by its index. We are able to extract the index from the space. These indices form a set, and we use the index set to store them. Each position of the array can have only one unique index; therefore, in the set, there are no duplicate entries, and the order of written the index is immaterial since the order of a index is carried by its value.

An index could be a non-negative integer (1-D array) or a tuple of $N$ non-negative integers for an $N$-D array, denoted such as $[3, 2, 5, \ldots]$. The values of indexes need not be sequential integers due to the existence of empty positions in the sparse array. Each

---

Figure 6.5: 1-D (a), 2-D (b), and 3-D (c) variation space
CHAPTER 6. DESIGN OF THE FORMALISM

The Space: \{[1]: [ ... ], [2]: [ ... ], [3]: [ ... ]\}

Index set: \{1,2,5\}

The Space: \{[0,0]: [ ... ],[0,2]: [ ... ],[0,4]: [ ... ]

\{[0,0]: [ ... ],[0,2]: [ ... ],[0,4]: [ ... ]

\{[0,0]: [ ... ],[0,2]: [ ... ],[0,4]: [ ... ]\}

Index set:\{[0,0],[0,2],[0,4],[1,0],[1,3],[1,5],[2,2],[2,4]\}.

Figure 6.6: Sparse variation spaces - 1D and 2D, and their index sets

...element in the index set associates with one element in the array at the position of the
element value. Figure 6.6 also shows the index sets for the two sparse spaces.

6.3.6 Distance of Two Variation Heads In a Space

Since the position (index) of a variation head may reflect its importance (or user’s preference), we should be able to measure the distance between two variation heads in a space. The use of properties’ values for measurement may not work because these values may not be continuous (these value could be sets, strings, or a series of numbers with uneven intervals). Since the index is made from integers, we can use the distance of two indexes to approximate the distance between two variation heads. For example, for two heads in a 1-D space, the distance of \(V[0]\) to \(V[1]\) is certainly closer than that of \(V[0]\) to \(V[3]\). Generalized to variation heads in a \(N\)-D space, the distance can be calculated by the following equation:

\[ d(V, W) = \|\text{index}_V - \text{index}_W\| = \sqrt{\sum_{i=1}^{n}(\text{index}_V - \text{index}_W_i)^2} \]
6.3.7 Some Primitive Operations Over a Variation Space

Since a variation space can be seen as an array of variation heads, there are several simple primitive operations that can be applied to a variation space. The order of variation heads in a space is highly intensional; the user should have full control to re-organize the space, such as to re-order, remove duplicated elements, etc. Similar to an array, we should be able to assign a variation head to a position, remove a variation head from a position (leave the space empty), or replace a variation head with another variation head. Since in our variation space the order is intentionally created by the user, it is important to give the user the ability to re-order or shift the position.

Index Mapping

*Index Mapping* changes the position (index) of a variation head through a mapping function over its index. Such a mapping function could be a function like \( I_{\text{new}} = I_{\text{old}} + n \) to move all heads back for \( n \) positions. Or it could be a list of mapping tuples like \( \{1 \Rightarrow 2, 3 \Rightarrow 4, \ldots\} \) to move individuals to new positions. To remove an empty position we can move higher indexed elements forward, or we can create empty locations by moving elements to a higher index. The positions of two variation heads can be swapped as well. Such index mapping can be done at any dimension in high dimension spaces (Figure 6.7).

For a given variation space \( V \) and a mapping function \( f_{\text{map}} \) (or a list of mapping tuples), the index mapping function will return a new space \( W \) as:

1. For each index \( i_V \) in \( \text{indexSet}_V \):
2. \( i_{\text{new}} = f_{\text{map}}(i_V) \)
3. \( W[i_{\text{new}}] = V[i_V] \)

In term of time complexity, if the function \( f_{\text{map}} \) is \( O(1) \), above index mapping algorithm is \( O(n) \).

6.4 Unification of Variation Spaces

Here we define *unification* as the operation to combine two or more spaces and write it as \( A \bigcup B \). Similar to unification that creates a new variation head by combining information
carried by the two variation heads, two variation spaces can be unified to create a new variation space that contains new variation heads. Such a process can be done through the unification of elements in one space with elements in the other space. Similar to the replication options of GenerativeComponents, we start with the following two methods of unifying linear variation spaces: Index Unification and Cartesian Unification.

### 6.4.1 Indexing Unification

Here we define *indexing unification* to create a new space where each element is the unification of elements at the same index of variation spaces. Elements inside two ordered variation spaces may have a one to one match relation over their index. For example, one variation space contains single property heads of one property, the other space contains single property heads of another property, and two variation heads at the same position represent two different properties of the same state. Through index unification, a new space containing more informative variation heads (with values in two properties) will be
generated (Figure 6.8, nodes’ names are omitted).

$$\begin{array}{ccc}
\downarrow & & \\
\downarrow & & \\
\end{array}$$

Figure 6.8: Unify two 1-D spaces by matching indices

Indexing unification can also work for $N$-dimensional spaces by following a similar rule of matching the index of variation heads from each dimension.

Index matching tries to unify elements within the same location in two spaces. Indexing unification does not require that the two operands have the same dimension and size. For two spaces with different dimensions, we have to convert the two spaces to the same dimensions. The lower dimensional space should be converted to match the higher dimensional ones by adding new dimensions. The length of the new added dimension(s) is 1.

Due to the existence of null (empty cell) in the space, we treat null as a special $\bot$ and define its operations as:

\[
\begin{align*}
null \sqcup \text{Object} &= \text{Object} \\
null \sqcup null &= null \\
null \sqcup \bot &= \bot \\
null \sqcup \top &= \top
\end{align*}
\]

There are six possible ways by which to unify different size spaces based on the operation of two index sets of the operands:

- **Intersection match** (written as $U \sqcap V$): The index matches only the overlapping part. The index set of the resulting space is the intersection of the operands’ index sets. The steps to unify two spaces $U$ and $V$ into $W$ in intersection match are:

1. $\text{indexSet}_W = \text{indexSet}_U \cap \text{indexSet}_V$.
2. For each index $i$ in the $\text{indexSet}_W$:
• **Union match** (written as $U \sqcup_{U} V$): The result covers the index from both operands. If one space does not have an element matching index with the other space, we automatically fill in a $\bot$ in the space. Since the result of a head unification with the bottom head is the head itself, when there is no matching index, essentially it is equivalent to copying over the elements onto the same index in the final result. The index set of the resulting space is the union of operands’ index sets. The steps to unify $U$ and $V$ into $W$ in the extended match are:

1. $\text{indexSet}_W = \text{indexSet}_U \cup \text{indexSet}_V$
2. For each index $i$ in the $\text{indexSet}_W$:
   $W[i] = U[i] \sqcup V[i]$

• **Left match** (written as $U \sqcup_{L} V$): The result has the index from the first (left) operand. If in the second space, there is no element in that space, use $\bot$ instead. The steps to unify $U$ and $V$ into $W$ in the left match are:

1. For each index $i$ in the $\text{indexSet}_U$
2. $W[i] = U[i] \sqcup V[i]$

• **Right match** (written as $U \sqcup_{R} V$): The result has the index from the second (right) operand. If in the first space there is no element in the space, use $\bot$ instead. To unify $U$ and $V$ into $W$ in the right match,

1. For each index $i$ in $\text{indexSet}_V$
2. $W[i] = U[i] \sqcup V[i]$

• **Left difference** (written as $U \sqcup_{LD} V$): The result has all of the variation heads in the first (left) space where the index does not exist in the second space. To get the left difference of $U$ to $V$ into $W$,

1. Let $\text{indexSet}_W = \text{indexSet}_U - \text{indexSet}_V$
2. For each index $i$ in the $\text{indexSet}_W$:
   $W[i] = U[i]$. 
• **Right difference** (written as $U \sqcup_{RD} V$): The result has all of the variation heads in the second (right) space where the index does not exist in the first space. $U \sqcup_{RD} V = V \sqcup_{LD} U$. To get the right difference of $U$ to $V$ into $W$,

1. Let $\text{indexSet}_W = \text{indexSet}_V - \text{indexSet}_V$
2. For each index $i$ in $\text{indexSet}_W$:

These operations can be demonstrated as Figure 6.9.

![Figure 6.9: Six ways of indexing unification](image)

**Properties**

Below are some properties of indexing unification:

$U \sqcup_i V = V \sqcup_i U$
$U \sqcup_U V = V \sqcup_U U$
$U \sqcup_L V \neq V \sqcup_L U$
The time complexity of all these unification methods will be $O(1)$.

**Append/Merge**

Combined with *index mapping*, index unification can merge two spaces into one or append one space behind the other. For a 1–D space, the append operation places each element from one variation space in order at the end of the other variation space. This is especially useful when we want to combine two single property spaces containing different variation heads into one bigger space (e.g., Figure 6.11).

![Diagram](image)

*Figure 6.10*: Index shifting and index unification to append two 1-D spaces

**Order Type Unification**

In previous sections, the unification of two values returns to the top if they are not equal. With more semantic meaningful order types involved (Section 6.2.2), the unification result may be more colorful. For example, unification of two ascending numbers returns the larger number. Within such unification relationships, unification of two spaces may have a richer set of variation heads.

Figure 6.11 shows the unification of spaces with different order types.
CHAPTER 6. DESIGN OF THE FORMALISM

6.4.2 Cartesian Unification

We define Cartesian unification \( (A \sqcup \otimes B) \) as the operation to create a new space in which elements are the unification result of all possible combinations of elements from input variation spaces. Through Cartesian unification, lower dimension spaces can create higher dimension spaces. Cartesian unification of \( N \) single property spaces will create a \( N \)-dimensional space.

 Cartesian Unification of One Variation Head and a Space

The Cartesian unification of one head with a space unifies the variation head with each variation head in the space. The result is a space that has exactly the same dimension and size as the input space. Using an index set, the procedure of Cartesian unification of variation head \( v \) with a space \( U \) to create a new space \( W \) can be described as:

1. For each index \( i \) in the index set of \( U \):
   
   \[ W[i] = U[i] \sqcup v \]

   \( W \) has the same dimension and size as \( U \).

   This one to space unification is commutative. \( v \sqcup U = U \sqcup v \) because unification is commutative: \( U[i] \sqcup v = v \sqcup U[i] \). The time complexity is \( O(N) \). Unification of two heads require constant time. The time of loops is the same as the size of the variation space.

 Cartesian Unification of Two 1-D Spaces

For two 1-D spaces \( U \) and \( V \), \( U = \{U_1, U_2, \ldots U_n\}, V = \{V_1, \ldots V_n\} \). \( U \sqcup \otimes V \) is a 2-D array. Based on the single head to space unification, the procedure is as follows: For each head
u (with index \(i_u\)) in \(U\), compute the Cartesian unification of the single head \(u\) with the space \(V\). Using index sets, \(U \uplus V \implies W\) can be done in a nested loop:

1. For each index \(i_U\) in \(\text{indexSet}_U\):
2. For each index \(i_V\) in \(\text{indexSet}_V\):
3. \(W[i_U, i_V] = U[i_U] \uplus V[i_V]\)

\(W\) is a 2-dimensional space. \(i_U, i_V\) are integers (or a single element tuple). The tuple \([i_U, i_V]\) is the index for the unified element. Since this operation uses a nested loop, the time complexity is \(O(m \times n)\), where \(m\) and \(n\) is the size of the two spaces.

The resulting 2-dimension space can be seen in Figure 6.12.

6.4.3 Cartesian Unification of Two High-Dimension Spaces

More generally, two spaces make a higher-order space through Cartesian unification. For two spaces \(U\) and \(V\), where \(U\) has a dimension of \(D_U\) and \(V\) has a dimension of \(D_V\), \(U \uplus V\) will have a dimension of \(D_U + D_V\). The general procedure is exactly the same as with two 1-D spaces: For each head \(u\) (with an index \(i_u\)) in \(U\), there is a Cartesian unification of \(u\) with \(V\).

Without loss of generality, the procedure of Cartesian unification of two high dimension spaces can be demonstrated by two 2-D spaces (Figure 6.13).

Using index sets provides us with a more clear way to describe the process. For \(U \uplus V \Rightarrow W\), \(\text{indexSet}_U, \text{indexSet}_V\), and \(\text{indexSet}_W\) are index sets for these three spaces. The nested loop solution from the above 1-D space unification can also be applied here. But since the index \(i_U\) and \(i_V\) are tuples of integers, the resulting index \(i_W\) is a longer tuple that combines all elements in \(i_U\) and \(i_V\) in sequence.

Also, the procedure can be illustrated with the following example:
\( UV = \text{indexSet}U \times \text{indexSet}V = \{(i, j) | i \in \text{indexSet}U, j \in \text{indexSet}V\} \)

Create a Cartesian product of \( \text{indexSet}U \) and \( \text{indexSet}V \). This result \( UV \) contains a set of all possible ordered pairs where the first element is a index from \( \text{indexSet}U \) and where the second element is a index from \( \text{indexSet}V \).

For each ordered pair \((i, j) \in UV\):

- \( k = \text{append}(i, j) = [i_0, \ldots, i_m, j_0, \ldots, j_n] \) // create a new index \( k \) that combines integer tuples in \( i \) ([i_0, \ldots, i_m]) and \( j \) ([j_0, \ldots, j_n]):

- \( W[k] = U[i] \sqcup V[j] \) // \( W \) is a \( M+N \) dimension space. \( i_U \) and \( i_V \) as indexes are tuples of integers. Combining them into the long tuple \( i_U, i_V \) is the index of unified elements.

In the sections below, I simply use \( i_W = [i_U, i_V] \) to denote combining two indexes \( i_U \) and \( i_V \) that place every element in \( i_U \) and elements in \( i_V \) into \( i_W \) in order.

![Cartesian Unification on two 2-D spaces](image)

**Figure 6.13:** Cartesian Unification on two 2-D spaces

The following provides a real example of the Cartesian product of two spaces (the unification result is \( \top \) if the same property has two different values) (Figure 6.14).
6.4.4 n-ary Cartesian Unification

Cartesian unification can be generalized to n-ary Cartesian unification over n variation spaces $S_0, S_1, \ldots, S_n$:

$$ R = S_0 \biguplus S_1 \biguplus \ldots \biguplus S_n $$

The whole process is quite similar to the Cartesian unification of two spaces. In the resulting variation space, each element $r$ is the unification result of elements $s_0 \bigcup s_1 \bigcup \ldots \bigcup s_n$ from all input variation spaces. The index of an element $I_r$ is composed from the index of each inputting element $[i_0, i_1, \ldots, i_n]$. 

The general algorithm of getting $R$ from $S_0, S_0, \ldots, S_n$ is:

```plaintext
1. $R = S_0$ // initiate
2. For $i=1; i<n; i++$
3. $R = R \biguplus S_i$ // Iteratively Cartesian Unification
```
CHAPTER 6. DESIGN OF THE FORMALISM

6.4.5 Properties

Cartesian product is neither commutative nor associative. Cartesian unification is not commutative, but it is associative.

\[ U \odot V \neq U \odot V, \] since the Cartesian unification creates higher dimensional spaces, and the order of the dimension (and the order of the tuples in the index) is associated with the order of operands in the Cartesian unification. For \( U \odot V \Rightarrow W \), \( w \)'s index will be \([i_U, i_V]\), while for \( V \odot U \), the resulting element’s index will be \([i_V, i_U]\).

\[(U \odot V) \odot W = U \odot (V \odot W).\] To approve:

- Unification is associative:
  \[ U[i_U] \sqcup (V[i_V] \sqcup W[i_W]) = (U[i_U] \sqcup V[i_V]) \sqcup W[i_W]. \]

- The combining index is associative: \([i_U, [i_V, i_W]] = [[i_U, i_V], i_W] = [i_U, i_V, i_W]\) since this operation put several tuples of integers into one in sequence.

Size of the Resulting Space

If there are \( N \) single property spaces involved, with the size of \( m_0, \ldots, m_n \) being the number of variation heads, then the final size of the resulting space will be \( m_0 \times \ldots \times m_n \), which may be a large number. If the order of the variation heads is not important, and if duplicate heads can be removed, we are able to reduce the size (especially when there are many duplicate heads). To do so, we convert the ordered single property space into an unordered set, remove duplicates, then recount to an ordered set in any order. To remove duplicates, we can order these variation heads through a comparison method, then check neighboring elements and remove duplicates. By doing so, variation heads in the final space automatically carry the order of the given comparison method.

6.4.6 Cartesian Square and Cartesian Power

A variation head could be a partial of the maximal variation head. One variation space may contain a list of such partial variation heads, with each variation head represents a different part (e.g., different properties) of the maximal variation head. We define Cartesian square unification or Cartesian power unification to assemble such a variation space to unify partial variation heads into more complete objects. A Cartesian square will try to unify any two partial objects, while the power unification will unify \( n \) partial objects.
In mathematics, the Cartesian product of a set $X$ with itself is known as a Cartesian square ($X^2 = X \times X$). The more general form, called Cartesian power, is a $n$-ary Cartesian product of a set $X$ as:

$$X^n = X \times X \times \ldots \times X = \{(x_1, \ldots, x_n) | x_i \in X \text{ for all } 1 \leq i \leq n\}.$$ 

Similarly, for a given variation space $S$, we can define its Cartesian square unification $S^{\otimes 2}$ and Cartesian power unification $S^{\otimes n}$ as the Cartesian unification of $S$ itself. In the resulting variation space of Cartesian power unification, each variation head contains information from two variation heads in the original space. For $n$ Cartesian power unification, the resulting variation head will contain information from the $n$ variation head of the original space.

$$R = S^{\otimes 2} = S \biguplus S$$
and

$$R = S^{\otimes n} = S \biguplus S \biguplus \ldots \biguplus S$$

Since the unification of two heads is commutative, the resulting heads from the Cartesian square and power unification have some unique features over the regular Cartesian unification:

$$R[i, i] = S[i] \sqcup S[i] = S[i]$$

$$R[m, n] = R[n, m] = S[m] \sqcup S[n] = S[n] \sqcup S[m]$$

For Cartesian power,

$$R[i, i, \ldots, i] = S[i] \sqcup S[i] \sqcup \ldots \sqcup S[i]$$

$$R[\ldots, m, \ldots, n] = R[\ldots, n, \ldots, m]$$

For a space $S$ with length $a$ (has $m$ heads), although the Cartesian unification result of $S^{\otimes 2}$ has a length of $m^2$, there is only a $m \times (m - 1)/2$ number of unique results (Figure 6.15). For a $n$-power unification, the number of unique results is $m \times (m - 1) \times \ldots \times (m - n + 1)/n!$. To make the computation more efficient, we only need to calculate less than half of the unifications of the regular Cartesian unification. For a Cartesian power unification, this will save significant computing time.

To compute only the in-redundant heads, we can do the following for the Cartesian power unification of a 1-D space $S$ into $R$: 

---

**CHAPTER 6. DESIGN OF THE FORMALISM**

112
CHAPTER 6. DESIGN OF THE FORMALISM

sort index in the ascending order
For each index i in indexSet
For each index j in indexSet and j > i
    $R[i, j] = S[i] \sqcup S[j]$. 

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
<td>0,4</td>
<td>0, ...</td>
<td>0,m</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td>1, ...</td>
<td>1,m</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2,3</td>
<td>2,4</td>
<td>2, ...</td>
<td>2,m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3,4</td>
<td>3, ...</td>
<td>3, m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4, ...</td>
<td>4,m</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.15: Cartesian square unification of a 1-D space - indices of heads that need to be unified

If $S$ is a high dimension space, then the index sorting and comparing can be done by comparing each dimension from the first dimension to the last dimension. For two indices $I = [i_0, i_1, \ldots, i_n]$ and $J = [j_0, j_1, \ldots, j_n]$, we can use the following function to compare if $I > J$.

function compareTwoIndex($I$, $J$)
    $n = \text{count}(I)$ // get length of the index
    For $k = 0; k < n; k++$
        if $i_k > j_k$ { return true; }
    return false;

For a Cartesian power unification, we can expand the above Cartesian power procedure into multiple loops as following (as an example of 3-ary unification):

Sort index in the ascending order
For each index $i$ in indexSet
    For each index $j$ in indexSet and $j > i$
        For each index $k$ in indexSet and $k > j$
            $R[i, j, k] = S[i] \sqcup S[j] \sqcup S[k]$

Power unification is an iterative process of unification with $S$, $S_{\sqcup \ldots}^{n+1} = S_{\sqcup}^{n} \sqcup S$. Each round of the Cartesian unification may create new variation heads. But when $n$ goes up,
there eventually won’t be any new variation head generated. The index can still grow, but these new indices will produce only repeated variation heads. We call such a n-power unification a fully grown unification. In an extreme case, each variation head in \( S \) represents a partial piece of one full head. In such a case, if the variation space \( S \) has a length of \( m \), we maximally need \((m - 1)\) power unification to get a space that has at least one variation head as the unification result of all heads in \( S \).

6.5 Spaces with Multiple-Property Variation Heads

Multiple-property variation heads can be produced through Cartesian unification and put into a high-dimension space. A series of multiple-property heads may be put into a list as a 1-D space. The user may know a number of valid variations of the graph (the explicit space that has been discovered by the user)(Figure 6.16). These variations define possible relations among the differing variation properties. In future computations, these relations may be preserved.

\[
\begin{bmatrix}
  p : 1 \\
  q : 1 \\
  r : 2 \\
\end{bmatrix}, \quad \begin{bmatrix}
  p : 1 \\
  q : 2 \\
  r : 2 \\
\end{bmatrix}, \quad \begin{bmatrix}
  p : 2 \\
  q : 1 \\
  r : 2 \\
\end{bmatrix}, \quad \begin{bmatrix}
  p : 3 \\
  q : 1 \\
  r : 3 \\
\end{bmatrix}, \quad \begin{bmatrix}
  p : 3 \\
  q : 2 \\
  r : 3 \\
\end{bmatrix}
\]

Figure 6.16: A 1-D space with multiple-property heads

Since a variation head contains pairs of property and values, we can look at it in two ways. One way is to think the pairs of values as constraints. A variation space contains a list of constraints. If one property is bounded with one value, then other properties can only hold a value from any of the given variation heads in the space. In the example of Figure 6.16, when property \( p \) is assigned with value 1, then \( q \) can only be either 1 or 2, and \( r \) can only be 2. The second way is to think a variation head describes a partial object. A maximal variation head describes all properties of the object, while a variation head may just know part of all properties. Then a variation space becomes a collection of partial objects.

From this point of view, Cartesian square and Cartesian power unification is especially useful for a multi-property space because they provide us a way to assemble partial objects or constraints. A fully grown variation space generated from Cartesian power
unification contains all possible completed objects or variation heads that satisfy relations and constraints carried by the initial variation space.

6.5.1 Orthogonal Space

If a high-dimensional space is constructed through the Cartesian unification of several single property spaces where each space has a unique variation property, we name such a space an orthogonal space. In an orthogonal space, each dimension represents one property of the head. Every element has exactly the same properties. These dimensions are orthogonal. When an index changes along one dimension, there will only be values changed along the same property.

A given graph can have a given number of variation properties (say $N$ properties), which can create $N$ single property variation spaces. If the Cartesian unification involves all of its $N$ single property spaces, then this $N$-dimensional orthogonal space covers all of the possible solutions/designs of the problem. Each element in this space is a maximal variation head.

The Cartesian unification resulting from a 1-D multiple-property space normally will not be an orthogonal space (Figure 6.17).

\[
\begin{array}{c}
\left[ \begin{array}{c} p:1 \\ q:2 \end{array} \right] & \left[ \begin{array}{c} p:2 \\ q:6 \end{array} \right] & \left[ \begin{array}{c} p:1 \\ q:3 \end{array} \right] & \uplus & \left[ \begin{array}{c} r:5 \end{array} \right] & \left[ \begin{array}{c} p:1 \\ r:9 \end{array} \right] \\
\downarrow \\
\left[ \begin{array}{c} p:1 \\ q:2 \\ r:5 \end{array} \right] & \left[ \begin{array}{c} p:2 \\ q:6 \\ r:5 \end{array} \right] & \left[ \begin{array}{c} p:1 \\ q:3 \\ r:5 \end{array} \right] & \uplus & \left[ \begin{array}{c} p:1 \\ q:3 \\ r:9 \end{array} \right] & \left[ \begin{array}{c} r:9 \end{array} \right] \\
\end{array}
\]

**Figure 6.17:** Cartesian unification of multiple-property heads. Values unify to $\top$ if two values are not equal.
Decomposition of an Orthogonal Space

Decomposition turns an orthogonal N-dimension variation space into its original single property spaces (Figure 6.18). If a space $V$ is the Cartesian unification result of $N$ single property spaces of unique properties, $V = V_1 \bigcup \otimes V_2 \bigcup \otimes \ldots V_n$, then the decomposition of $V$ is a list of single spaces $\{V_1, V_2, \ldots, V_n\}$.

![Diagram of Decomposition of a 2-D space](image)

Figure 6.18: Decomposition of a 2-D space

For orthogonal high dimensional spaces, the simplest way is to decompose both input spaces into sets of single property spaces and then to create a new Cartesian unification from these single property spaces. These duplicated dimensions/properties may need special attention depending on the unification rule; the Cartesian product of two spaces with the same dimension may raise many $\top$.

### 6.5.2 Projecting a Multi-Property Space into Single Property Spaces

We have defined the function

\[ \text{propertyV}(G, p) \]

to return one variation property of a graph. By applying this function onto each of the elements in the above multi-property space, we are able to generate a space with single property variations with the same dimension and size.

*Projection* is the operation which projects a multi-property space into another space with selected properties.
The projection result of $\top$ is $\top$, and the projection of $\bot$ is $\bot$.

Figure 6.19 are projection results of the resulting space of Figure 6.17. Each result space contains only one property.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p:1$</td>
<td>$q:1$</td>
<td></td>
</tr>
<tr>
<td>$p:2$</td>
<td>$q:2$</td>
<td></td>
</tr>
<tr>
<td>$p:3$</td>
<td>$q:1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p:1$</td>
<td>$q:2$</td>
<td>$p:3$</td>
</tr>
<tr>
<td>$p:2$</td>
<td>$q:2$</td>
<td>$p:3$</td>
</tr>
<tr>
<td>$p:3$</td>
<td>$q:2$</td>
<td>$p:3$</td>
</tr>
</tbody>
</table>

Project $\downarrow$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p:1$</td>
<td>$p:2$</td>
<td>$p:3$</td>
</tr>
<tr>
<td>$q:1$</td>
<td>$q:2$</td>
<td>$q:3$</td>
</tr>
<tr>
<td>$q:1$</td>
<td>$q:2$</td>
<td>$q:3$</td>
</tr>
<tr>
<td>$r:5$</td>
<td>$r:5$</td>
<td>$r:9$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.19: Projection of the resulting space of Figure 6.17 to single properties

To maximally maintain the information carried by the original space, each of the projected single property spaces must have the same order and size as the original space. Through indexing unification (6.4.1), these single property spaces can be unified to form the initial multi-property space. There may exist many duplicated values in each projected space. If the order in the space is not important, we can remove duplications and assign new indices to remaining variation heads.

The Cartesian unification result produced may not be decomposed into the input original spaces through projection. Due to unification, the result may contain $\top$, which loses input information.

### 6.5.3 Flattening a High-Dimension Space

A high-dimension space may be simplified to a simple set by removing duplicates and dimensions. We can choose to remove or keep $\top$ and $\bot$. The operation is as simple as:
6. Design of the Formalism

Set up an empty 1-D space $H$. For each variation head $A$ in the existing $N$-dimension space $G$, if $A$ does not exist in $H$, append $A$ to $H$.

The flattened results of Figure 6.19 are shown in Figure 6.20.

```
\[
\begin{array}{c|c|c}
  \text{p:1} & \text{p:2} & \text{p:1} \\
  \text{p:1} & \text{T} & \text{p:1}
\end{array}
\]
```

Flatten ↓

```
\[
\begin{array}{c|c}
  \text{p:1} & \text{p:2}
\end{array}
\]
```

**Figure 6.20:** Flattening one result space of figure 6.19

### 6.5.4 Unify Variation Heads Along One Dimension

We may unify variation heads of a space along one (or more) dimension(s). The resulting space will be a space with reduced dimension(s). For a 1-D space, this procedure will make one single variation head by unifying all variation heads in the space. For a 2-D space, unification along one (the first or the second) dimension will cause a 1-D space. The removed dimension(s) is the one we choose to unify.

In an N-dimensional space $S$, a variation head has an index like $[i_0, i_1, \ldots, i_m, \ldots, i_{n-1}]$. The length of the $m_{th}$ dimension is $L$, where the valid index numbers at the $m_{th}$ dimension are $p_0, p_1, \ldots, p_{L-1}$. To unify elements along the $m_{th}$ dimension (the length of the $m_{th}$ dimension is $L$), we can:

1. Create an empty $N-1$ dimension space $T$.
2. For each index $I$ in $S$
   - $I = [i_0, i_1, \ldots, i_m, \ldots, i_{n-1}]$
   - $I_r = [i_0, i_1, \ldots, i_{m-1}, i_{m+1}, \ldots, i_{n-1}]$ // create the reduced index $I_r$ by removing $i_m$
3. if $I_r$ does not exist in $T$
4.   // Find all other variation heads that only differ on the index at the $m_{th}$ dimension:
   - $I_0 = [i_0, i_1, \ldots, p_0, \ldots, i_{n-1}]$
   - $I_1 = [i_0, i_1, \ldots, p_1, \ldots, i_{n-1}]$
   - $\ldots$
   - $I_L = [i_0, i_1, \ldots, p_{L-1}, \ldots, i_{n-1}]$
$v = S[J_0] \sqcup S[J_1] \sqcup \ldots \sqcup S[J_L]$ // Unify these variation heads to make a new head $v$

$T[I_r] = v$ // Assign $v$ into $T$

Repetition of the above procedure allows us to unify along more than one dimension.

\[
\begin{bmatrix}
p: 1 \\
q: 2 \\
r: 5 
\end{bmatrix}
\begin{bmatrix}
p: 2 \\
q: 6 \\
r: 5 
\end{bmatrix}
\begin{bmatrix}
p: 1 \\
q: 3 \\
r: 5 
\end{bmatrix}
\begin{bmatrix}
p: 1 \\
q: 2 \\
r: 9 
\end{bmatrix}
\begin{bmatrix}
p: 2 \\
q: 5 \\
r: 9 
\end{bmatrix}
\begin{bmatrix}
p: 2 \\
q: 6 \\
r: 5 
\end{bmatrix}
\]

Unify along the 2nd dimension $\downarrow$

\[
\begin{bmatrix}
\top \\
\begin{bmatrix}
p: 2 \\
q: 6 \\
r: 5 
\end{bmatrix}
\end{bmatrix}
\top
\]

**Figure 6.21**: Unify along the 2nd dimension (column) of a space

This operation may create many $\top$ (Figure 6.21). But this could be useful for some order types like the ascending (descending) number, where the unification of many values can lead to the maximum (minimum) number. For example, in figure 6.22, where the properties’ values are ascending integers, the relation of subsumption and unification is:

- $a \sqsubseteq b$ if $a \leq b$ or $a = \bot$ (e.g., $2 \sqsubseteq 3$)
- $a \sqcup b = b$ if $a \leq b$ (e.g., $2 \sqcup 3 = 3$)

Then the result of unifying heads in this 1-D space is a variation head with a maximum value in its properties.

\[
\begin{bmatrix}
p: 1 \\
q: 2 
\end{bmatrix}
\begin{bmatrix}
p: 2 \\
q: 6 
\end{bmatrix}
\begin{bmatrix}
p: 3 \\
q: 3 
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
p: 3 \\
q: 6 
\end{bmatrix}
\]

**Figure 6.22**: Unify 1-D space carrying values of ascending numbers
6.5.5 The Hysterical Space

Several Ways of Appending High-Dimension Spaces

Once we are given two high-dimension spaces, other than direct indexing unification and Cartesian unification, the most direct way is to append one space after the other. Such append operation can be done in two steps: first, use index mapping to insert empty slots in one or several dimensions in one space. Second, use indexing unification to unify the other space with this new space. However, the way to choose which dimension(s) to insert the empty slot will create different result. Figure 6.24, 6.25, and 6.26 demonstrate three different ways of appending space A and B of Figure 6.23. Figure 6.24 append B at the end of A's first dimension. To do so, at first we use indexing shifting to move indices of B backward to create the space $B_1$, then union unify $B_1$ with A. With a similar method, Figure 6.25 append B at the end of second dimension. Figure 6.26 append B at end of first and second dimension. However, in Figure 6.26 maybe better than the other two because it is an orthogonal space. Although both A and B are orthogonal spaces, the results in Figure 6.24 and 6.25 are not orthogonal. Indices and property values are not corresponding in these two result spaces. Figure 6.26 is orthogonal, but in the result there are many empty spaces. Is it possible to fill in these spaces from the known spaces?

![Figure 6.23: How to combine these two spaces?](image)

Generate the Hysterical Space

Woodbury et al. (2000) proposes the concept of design hysteresis to describe the idea of constructing implicit space states by erasing and recombining explicit space states. The implicit space contains all possible states, while the explicit space comprises those states.
that have been visited during the design process (Woodbury and Burrow, 2006). Sheikholeslami (2009) proposes the term hysterical state and hysterical space. A hysterical state can be reached by recombining prior variable settings in the parametric design. The hysterical space describes the set of states that can be reached through such variables recombination. He proposes using Cartesian product of variable settings to construct the hysterical space. Sheikholeslami (2009)'s solution can be implemented through our variation space representation.

Suppose we have $n$ variation spaces $V_1, \ldots, V_n$. To compute the hysterical space $H$: 
CHAPTER 6. DESIGN OF THE FORMALISM

![Figure 6.26: Append B at the end of both dimensions of A](image)

\[
\begin{bmatrix}
  p:1 \\
  q:1 \\
\end{bmatrix} \quad \begin{bmatrix}
  p:1 \\
  q:2 \\
\end{bmatrix}
\]

\[
\Rightarrow
\begin{bmatrix}
  p:3 \\
  q:3 \\
\end{bmatrix} \quad \begin{bmatrix}
  p:3 \\
  q:4 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  p:4 \\
  q:3 \\
\end{bmatrix} \quad \begin{bmatrix}
  p:4 \\
  q:4 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  p:2 \\
  q:1 \\
\end{bmatrix} \quad \begin{bmatrix}
  p:2 \\
  q:2 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  p:3 \\
  q:3 \\
\end{batrix} \quad \begin{bmatrix}
  p:3 \\
  q:4 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  p:4 \\
  q:4 \\
\end{bmatrix}
\]

\[
B_3
\]

\[
Result_3
\]

1. \( V = V_1 \text{ append } V_2 \text{ append } \ldots V_n \) // Combine these spaces into one by append
2. \( W = \text{flatten}(V) \) // Flatten \( V \) to remove duplicate variation heads. \( W \) is a \( m \)-property space.
3. For each property \( i \) in \( m \) properties of \( W \) //Next we use projection to project \( W \) into \( m \) single property spaces.
4. \( W_i = W \text{ projection on } i \)
5. For each \( W_i \)
6. \( X_i = \text{flatten}(W_i) \) //use flatten to remove duplicate single property heads. Now we have \( m \) spaces as \( X_1, X_2, \ldots, X_m \).
7. \( H = X_1 \biguplus X_2 \biguplus \ldots X_m \) //The final hysterical space \( H \)

Thus, from the same input of Figure 6.23, we can obtain the hysteric space of Figure 6.27.

We do not need many spaces to create the hysterical space. We can create the hysterical space from one 1-D multi-property space such as Figure 6.16, or from several known variation heads. With several know variation heads, we can generate several single property spaces, which will yields a hysterical space.
This chapter described the basic design of the formalism. Instead of more general alternatives, this research focuses on variations of parametric model. In the formalism, the basic object is the variation head; it has two basic operations, subsumption and unification. A variation space is a collection of variation heads in the form of an indexed array. The index reflects the order of a variation head in a space, and the order may reflect the user’s intention. The index set of variation space plays an important role in the operations on variation spaces.

Variation spaces provide us with several operations. The two most important operations are indexing unification and Cartesian unification, which can unify two or more spaces into one. Through combination, these operations can give us more useful ways to deal with individual and multiple spaces.
Chapter 7

Relational Algebra View of the Formalism

Our variation space is analogous to the relational model. The process of unifying several variation spaces carries is similar to the idea of the join operator that combines several database tables. The relational model was first formulated and proposed by Codd (1969, 1990). It forms the foundation of the current popular relational database management systems (RDBMS). Relational algebra is a formal description of how to manipulate relations in the model. It is the mathematics that underpin modern SQL database operations and the interface to the data stored in a relational database. Relational algebra provides ideas to extend our formalism, especially our operations, on variation spaces.

7.1 The Notion of the Relational Model

Given sets $S_1, S_2, \ldots, S_n$, a relation $R$ is a set of $n$-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in S_i$. More concisely, $R$ is a subset of the Cartesian product of $S_1 \times S_2 \times \ldots \times S_n$. $S_i$ is called the $i$th domain of $R$. An $n$-tuple is called an element of the relation (Codd, 1969). Depending on the number $n$, a relation could be a binary relation (2-tuple), a ternary relation (3-tuple), or a $n$-ary relation.

To be more convenient to both the user and computing, a relation is composed of a heading and a body. Instead of using position ordering for the tuple in the element ($n$-tuple), domains should be identifiable by a unique name, which is called the attribute of
the relation R. A tuple of attribute values is called an element. The tuple of all attribute names is called a relationschema (or heading). Matching the heading’s attribute names to values in the body, the element can also be seen as \( n \) attribute/value pairs.

A relation can be represented as a table. Attributes become table columns. Each table column has a unique column name and only permits certain types (or values) of data. All unique values in one column form one domain. Rows of the table are the elements (n-tuples).

Normally one domain (or combination of domains) has a value to uniquely identify each element in the relation. Such a domain is called a primary key (Codd, 1969). A relation may have two or more primary keys. One of these primary keys will be chosen as the primary key.

Although values in domains are normally atomic, non-atomic values are valid in the relational framework (for example, value pairs, tuples, or sets). A relation can also cross-reference other elements (in the same relation or other relations) through a foreign key. A domain (or domain combination) of relations R is a foreign key if its values are the primary key of relation S (R and S may be identical).

Null is a special marker in the relational model to indicate that a value does not exist in the relation. This concept, extended by Codd in 1979, represents the “missing information and inapplicable information” (Codd, 1979). It requires special functions and predicates to handle Nulls, such as, “is NULL” to check if one value is NULL.

Figure 7.1 is a relation to store person information on a relation schema of (SinID, lastName, bornCity, gender). In this example, sinID is the primary key that identifies unique elements.

7.1.1 Map Variation Space to Relational Model

We have defined three objects in our variation space formalism: node with attributes, variation heads, and variation space. A variation head is notated as node.property/value pairs. Given an ordering of these properties, the variation head can be written as a tuple of values, with the values corresponding to the order of properties. A variation space is a collection of variation heads with indices, which can be seen as a set of tuples, with each tuple has an index as the first member, followed by the tuple of the variation head. If some variation heads miss some properties, we can insert \( \perp \) for these missing properties.
SinID = { 1311, 2205, 3234, 4421 }
lastName = { Lee, Smith, Lam }
bornCity = { Vancouver, Burnaby }
gender = { male, female }

<table>
<thead>
<tr>
<th>sinID</th>
<th>lastName</th>
<th>bornCity</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>1311</td>
<td>Lee</td>
<td>Vancouver</td>
<td>male</td>
</tr>
<tr>
<td>2205</td>
<td>Smith</td>
<td>Burnaby</td>
<td>male</td>
</tr>
<tr>
<td>3234</td>
<td>Smith</td>
<td>Null</td>
<td>female</td>
</tr>
<tr>
<td>4421</td>
<td>Lam</td>
<td>Burnaby</td>
<td>male</td>
</tr>
</tbody>
</table>

Figure 7.1: An relation example - 4 domains and the table view. Null means that piece of information is unknown/missing.

Based on Codd’s definition, we can map a variation space into the relational model (Codd, 1969). A variation space \( S \) has \( n \) variation properties, with each property having a set of values (plus \( \perp \) and \( \top \)). These values make \( n \) sets \( V_1, V_2, \ldots, V_n \). Since the index is essential in a variation space, all indices makes another set \( ID \). The variation space \( S \) is a subset of \( n + 1 \) tuples \( (id, v_1, v_2, \ldots, v_n) \), where \( id \in ID, v_i \in V_i \). More concisely, \( S \) is a subset of the Cartesian product of \( ID \times V_1 \times V_2 \times \cdots \times V_n \). In this definition, an index is a domain of the relation \( S \). All variation properties are also domains. An element of the relation is a \( n + 1 \) tuple, contains the index and \( n \) variation properties’ values. Such a relation has the same property as defined by Codd:

1. Each row represents an \( n + 1 \) tuple of \( S \).

2. The ordering of rows is immaterial because the ordering is already carried by the index.

3. All rows are distinct, because the index in the row is distinct.

4. The ordering of columns is significant - the ordering is responding to the order of variation properties \( V_1, V_2, \ldots, V_n \).

5. The significance of each column is partially conveyed by labeling it with the name of the corresponding variation property.

For a given variation space, we can simply map its index and all variation properties as the element headings of a relation (Figure 7.2). In this figure, the space has \( L \) variation heads with indices from \( index_1 \) to \( index_L \). The variation heads have many nodes (named A
### CHAPTER 7. RELATIONAL ALGEBRA VIEW OF THE FORMALISM

<table>
<thead>
<tr>
<th>index</th>
<th>$A.p_1$</th>
<th>...</th>
<th>$A.p_n$</th>
<th>...</th>
<th>$H.q_1$</th>
<th>...</th>
<th>$H.q_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>index</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 7.2:** A relation directly mapped from a variation space

$\{[0,0]:{A:{x=1, y=1}, B:{q=4}}, [0,1]:{A:{x=1}, B:{p=7, q=8}}, [1,0]:{A:{x=2, y=1}, B:{p=3}}, [1,1]:{A:{x=1, y=1}, B:{p=3}} \}$

<table>
<thead>
<tr>
<th>index</th>
<th>$A.x$</th>
<th>$A.y$</th>
<th>$B.p$</th>
<th>$B.q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1</td>
<td>1</td>
<td>$\bot$</td>
<td>4</td>
</tr>
<tr>
<td>0,1</td>
<td>1</td>
<td>$\bot$</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1,0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>$\bot$</td>
</tr>
<tr>
<td>1,1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

**Figure 7.3:** A relation view of a variation space

... and properties as $\{A.p_1, \ldots, A.p_n, \ldots, H.q_1, \ldots, H.q_m\}$. An example populated with values can be seen in Figure 7.3.

**Bottom $\bot$, Top $\top$ and Operators**

Relational algebra uses normal selection (binary operations in the set $\{<, \leq, =, \geq, >\}$) and logical operators of $\lor$ (or), $\land$ (and), and $\neg$ (negation) to compare and select elements by their values. Join operators are used to combine elements in two (or many) relations. In variation space, we use $\sqsubseteq$ to compare two values so that we may see if one value contains more information than the other, and we use $\sqcup$ to combine two objects into one.

From one perspective, $\bot$ in our formalism is similar to $\text{Null}$ in relational algebra. They both represent missing information. However, the way we handle $\bot$ in variation space is different than the way relational algebra handles $\text{Null}$ (Figure 7.4). In relational algebra, $\text{Null} \ op$ anything, where $\text{op}$ is a logical operator, results in $\text{Null}$. In our variation spaces, it is always true that $\bot$ subsumes a given value. Here we treat $\bot$ as a extension of $\text{Null}$. When dealing with one of the binary operations in the set of $\{<, \leq, =, \geq, >\}$, we can treat $\bot$ as $\text{Null}$ because $\bot$ provides no information to compare.

Top $\top$ in our variation space happens after unification $\sqcup$. It represents a state of “conflict” where information held by two operands is not unifiable. The unification result of anything with $\top$ is always $\top$. In relational algebra, if the join condition does not
match, the result is simply discarded. But in our variation space, the unification result is marked with $\top$, and it can be used in future operations (although the result is always $\top$). By keeping $\top$, we can actually account for these non-unifiable objects that may contain valuable information for us to use. $\top$ can be treated as a special value in the relation. For an element, if one attribute is $\top$, then the whole element is $\top$.

### 7.2 Relational Algebra

Relational algebra is an algebra constructed specifically to deal with relations. Its operators deal with one or more relations in order to yield a relation. Relational algebra has six primitive operators: selection, projection, Cartesian product (cross product or cross join), set union, set difference, and rename.

#### 7.2.1 Selection

In relational algebra, selection is a unary operation that selects all elements in the relation which have some relation(s) that match a given criteria. It can be written as $\sigma_\varphi(R)$, where $\varphi$ is a propositional formula, and $R$ is the relation. In more detail, it can be written as $\sigma_{a\theta b}(R)$ or $\sigma_{a\theta b}(R)$, where:

- $a$ and $b$ are attribute names
- $\theta$ is a binary operation in the set of $\langle, \leq, =, \geq, \rangle$.
- $v$ is a value constant
• $R$ is a relation
• $\sigma_{a \theta b}(R)$ selects all tuples in $R$ that $\theta$ holds between attribute $a$ and $b$
• $\sigma_{a \theta v}(R)$ selects all tuples in $R$ that $\theta$ holds between the attribute $a$ and the given value $v$.

In our variation space, the selection operation can give us all of the variation heads with the given nodes (and properties) that satisfy a propositional formula. However, the proposition formula is different. Currently we have two operations by which to compare two order types: subsumption ($\sqsubseteq$) and equality ($=$). We can use selection to select all elements in a space that satisfy a condition; for example, variation heads that have one node (or one property) subsume a given value, or variation heads that are subsumed to a given variation head.

In order to use binary operations like $<, \leq, \geq,$ or $>$, we need to convert order types into comparable atomic values. For example, as a ad-hoc way, we can take the integer number from a lifted integer object to participate in the binary operation.

### 7.2.2 Projection

In relational algebra, *projection* is a unary operation written as $\pi_{a_1, \ldots, a_n}(R)$, where $a_1, \ldots, a_n$ is a set of attribute names. It gives all tuples in $R$ that are restricted to attributes $(a_1, \ldots, a_n)$. In a database table, this operation can be thought of as picking several columns from all available columns.

We can use such a projection operator to create a space with projected variation heads from the original space. The projection can happen at the node relation level or at the variation head relation level. A projected node contains only a subset of properties of the original node. A projected variation head contains only a subset of nodes in the original head. Through projection, we can easily get a single property space from a multiple property space.

### 7.2.3 Rename

The *rename* operation is a unary operation over a relation. The result is a relation that is identical to the original except that one attribute has been renamed to another. In our parametric model and variation space, the node’s name is significant; it provides
the anchor point for unification and is essential in constraint expressions for linkages. Renaming provides us with flexible equipment by which to match variation heads onto different parametric models by renaming the nodes’ names or property names.

7.2.4 Set Operations

Set operators are taken from set theory. These operations require that the two relations are union-compatible; the two relations must have exactly the same set of attributes. Two variation spaces must have the same dimension to be union-compatible. In term of properties, in a variation space, if a node property is missing, its value is equivalent to ⊥. Thus, we can easily make two variation spaces union-compatible by assigning ⊥ to missing properties.

Set Union

In relational algebra, the union creates a relation that includes all tuples which existed in either (or both) input relations. Duplicated tuples are eliminated.

Due to the existence of index, we can not arbitrarily union two variation spaces together if the two spaces contains same indices because an index has to be unique. Our union indexing unification has the same effect. If two variation spaces do not have any duplicated index or the indices with associated variation heads are exactly the same, we can directly apply set union. If there exist the same indices, simply putting indices and variation heads into one space is not valid because this will cause duplicate indices. Our existing union indexing unification provides a good method by unifying variation heads with the same index.

Set Difference

The difference of relations R and S is a relation that contains all of the tuples that are in R but not in S.

Our existing left and right difference indexing unification is an analog to this set difference operation. The difference indexing unification of two variation spaces will create a new space that contains the index (and its associate variation head) which is in one space but not the other. Since ordering is important in our context, we provide two different
means of index difference unification - left difference and right difference. Left difference is similar to the operand ordering of set difference, while right difference reverse the operand ordering.

**Set Intersection**

*Intersection* is not listed as a primitive operator of relational algebra since it can be derived from union and difference.

The intersection of two relations gives a relation that includes all tuples that are in both inputting relations. The analog in variation space is intersection indexing unification. The intersection unification of two variation spaces gives a space that has the indices that appear in both inputting spaces. The associated variation heads of the indices are the unified result from both variation spaces.

### 7.3 Join vs. Unification

In relational algebra, join is the operation to combine two relations without loss information. A binary relation $R$ is joinable with a binary relation $S$ if there exists a ternary relation $U$ such that \( \pi_{12}(U) = R \) and \( \pi_{23}(U) = S \), which means, through projection, we can get the original input relations from the result relation.

Under variation spaces, Cartesian unification combines the variation heads from one space with all of the heads from the other space. The resulting space contains variations heads with information from both spaces, and the number of variation heads is the multiplication of heads from the two spaces. However, with order type unification, some information carried from the input variation heads may be lost. Unification combines information carried by the two input heads to make a new result head. But we are not able to retrieve the original two variation heads from the new head. There, the information carried by original head is lost. The simple example would be for two ascending number, \( A \sqsubseteq B = B \) if \( A \leq B \), \( A \sqcup B = B \). With the final result $B$ only, we can not retrieve the original value of $A$. Therefore, we can not directly use the join operator in our variation space. But we can still borrow this concept and consider our unification operations as analogs of join.

Cartesian product (also called cross join or cross product) combines tuples of one relation with all of the tuples from the other relation. The resulting relation combines
attributes from both relations, and the number of tuples is equal to the product of the numbers of tuples from the two relations. In relational algebra, the main join operators are natural join, \( \theta \) join, and outer join. These operators provide analogues to enrich our unification operators.

### 7.3.1 Natural Join, Indexing Unification, and Cartesian Unification

Natural join (\( \bowtie \)) is a binary operator over the two relations \( R \) and \( S \), and is written as (\( R \bowtie S \)). Natural join combines all tuples in two relations that are equal on their common attribute names. Rename may be used upon the operating relations since natural join uses the attribute name to match the tuple value.

Our variation space unification is analogous to natural join for joining pairs of index/variation head tuples. Subsumption always compares the common properties of nodes with the same name in variation heads. Since the index serves as the primary key in the variation space, the way we handle the index attribute makes a difference. If we require that the unification shall happen only if two indexes are the same, then the end result is akin to an intersection indexing unification.

The join operator tries to match all possible pairs of index/variation head tuples. However, since the value of the index is unique, then for a given index/variation head tuple in one space, there will only exist one (or no) index/variation tuple in the other space with the same index. Therefore, if index equality is required, the only possible index/variation head pairs to join are those pairs with the same index.

Our Cartesian unification can be seen as an analog of natural join without index equality. It can be done in steps: First, rename the common "index" attribute of two spaces (say \( \text{leftSpace} \) and \( \text{rightSpace} \)) into unique names (e.g., \( \text{leftSpace.index} \) and \( \text{rightSpace.index} \)). Second, apply natural join. Last, combine the two index attributes (\( \text{leftSpace.index} \) and \( \text{rightSpace.index} \)) to make the final index.

In our unification, if two variation heads do not unify, we can maintain the record with a value of \( \top \). Relational algebra’s join always discards Cartesian product pairs that do not satisfy the given condition. To remove \( \top \) from our unification result, we can explicitly apply an operation over the whole space to remove tops.
7.3.2 \( \theta \) Join and \( \theta \) Unification

\( \theta \) join is a more general form than natural join. The combination condition between two relations is not simply the equality of shared attributes. \( \theta \) join uses a binary operator as the join condition to compare an attribute to an attribute (or value) when combining tuples. The resulting relation contains all of the combinations of tuples that satisfy the join condition. It can be written as

\[
R \bowtie S \quad \text{or} \quad R \bowtie S
\]

\[
a \theta b \quad \text{or} \quad a \theta v
\]

where \( a \) and \( b \) are attribute names, \( \theta \) is a binary relation in the set \( \{<,\le,=,\ge,>\} \), \( v \) is a value constant, and \( R \) and \( S \) are relations. The result of this operation consists of all combinations of tuples in \( R \) and \( S \) that satisfy the relation \( \theta \).

We can use similar concepts in our variation space unification and call it \( \theta \) unification (Figure 7.5). In the original Cartesian unification and indexing unification, there is no restriction on which nodes are eligible to participate in the unification. Also the original unification method only compares the same properties of those nodes with the same name. With \( \theta \) unification, we can apply rules to compare values of different properties, such as, the property value of one node in one space subsumes a different node in the other space. For example, we have two spaces: \( R \) and \( S \). A variation head in \( R \) only contains one node, named \( A \), and a variation head of \( R \) contains another node, \( B \). For the
regular Cartesian unification, the unified space will have all of the possible combination
of variation heads that have \( A \) and \( B \). With \( \theta \) unification, we can apply a condition such
as \( A.\text{property}X \sqsubseteq B.\text{property}Y \) (these two properties must have the same type of values).
Then the resulting space will only have variation heads that have nodes that satisfy the
condition.

We can write \( \theta \) unification as:

\[
R \underleftarrow{\theta} S \\
\theta \quad a \theta b \\
r \underleftarrow{\theta} S \\
\theta \quad a \theta v
\]

where \( a \) and \( b \) are attribute names, \( \theta \) is the operation of \( \sqsubseteq \) or \( = \) (equality), \( v \) is an order
type, and \( R \) and \( S \) are variation spaces.

### 7.3.3 Outer Join and Outer Unification

Natural join and \( \theta \) join combine the tuples that exist in both operand relations. Sometimes
this may cause a loss of information since, for both, the source tuple must exist. An outer
join retains the information that would have been lost from the relations. Some of the
resulting tuples are formed by extending unmatched tuples by filling values (Null value
\( \omega \)) for every attribute of the other relation. There are three types of outer joins: full, left,
and right. In left outer join, the resulting relation always contains elements from the left
operand. The resulting relation from the right outer join always contains elements from
the right operand. The relation from full outer join will always contain elements from
both sides. If there are no matching elements from the other operand, these outer joins
will fill in Null values for these attributes which have no matching value.

In our indexing unification methods (other than intersection indexing unification),
the union indexing unification, left indexing unification, right indexing unification, left
difference unification, and right difference unification all use the concept of outer join on
indices: if the index (and, of course, the head associated to it) does not exist in one space,
we still keep the information (the index and associate variation head) from the other space.

Similarly, we can put the outer join concept into our \( \theta \) unification to make it outer
unification. If one space does not have any variation head that satisfies the condition, we
can use \( \perp \) instead to participate in the unification (Figure 7.6). In a regular \( \theta \) join, like the
\( R \) and \( S \) example in above \( \theta \) unification example, the resulting space only has variation
heads with two nodes \( A \) and \( B \) that satisfying the condition \( A.\text{property}X \sqsubseteq B.\text{property}Y \).
For left outer unification, the resulting space’s variation heads will use all of the variation heads from \( R \) (the left operand) to participate in the unification. If \( S \) does not have any variation head that satisfies the condition, the resulting space will simply be the variation heads from \( R \). Thus, the resulting space may have many variation heads that have only one node, i.e., the node \( A \). Similarly, for right outer unification, the resulting space may have variation heads that only have node \( B \). For full outer join, the resulting spaces’ head may only contain either node \( A \) or \( B \).

After Cartesian unification, the index of the resulting space is a compound index from both operands. In outer unification, if there is no matching head from one operand, there is no index either. For such case, we have to fill in a value (\( \perp \)) in the index to make sure the resulting index is consistent with others (Figure 7.6).

### 7.4 Summary

From several perspectives, we can see connections between our formalism and the relational algebra.

Our organization for variation spaces, index, and variation properties in variation heads can be mapped onto relations in relational algebra. A variation space can be seen as a relation. A variation head and its associated index in the space is a record (element) in the relation. Nodes are elements. Based on such a mapping, many operators from...
relational algebra can be adapted into our formalism. For example, set operators, rename, and projection. However, the join operators (e.g., natural join) in relational algebra can not be directly mapped with our unification since after unification, some original information may not be able to retrieve back. But still we can treat our unification operations as analogs of joins. The long established theory and practice of relational algebra will make our formalism more solid in terms of theory foundations and more rich for practical operations.
Chapter 8

Simulation of the Formalism

We have built concepts and algorithms for our formalism solely with pen and paper in the previous chapters. To better understand the characteristic and potential of these algorithms, we need to thoroughly test the formalism and explore these algorithms. Therefore, I developed a prototype system to simulate the formalism by computing and visualizing 3-D geometric data (Figure 8.1) so that we can visually analyze the input and output and make sense of variation spaces and operations over these spaces. Other than confirming and understanding this formalism, I also expected to use this prototype to iteratively improve and enrich operations and algorithms.

8.1 Design of the Prototype

This prototype directly implements the formalism including objects and algorithms. The main purpose of this prototype is to verify and enrich the formalism, though it does not deal with real problems. Within this prototype, we hope to find the limitations of the current proposed formalism when dealing with all possible combinations of input, and hence to generate new ideas to provide for missing concepts and operations.

Based on this prototype, we can understand more about the characteristics of the formalism, including the nature of high-dimensional spaces generated from the Cartesian unification, possible ways of index mapping, and operations of multiple-property heads spaces. We are interested in both the values of variation heads and the index of a variation head in its space.

The values of variation heads in a space are certainly important because these values
will participate in the symbolic model so that the model can compute successor node values. These values determine the state of the system. The indices of variation heads are equally important as their values. At the time when the user initializes a variation space (normally a 1-D space), indices may be intentionally assigned by the user to reflect an order that indicates its importance (or the user's preference). A compound index generated from unification will inherit the order information from input spaces. As the product of Cartesian unification, there may exist many results indicated as $\top$. The distribution of indices of non-top results are useful for our understanding of the effects of unification, and it may be helpful to direct our future use when navigating through the indices.

The visual inspection of values and indices may also help us to better understand values and indices of a variation space, as well as the Cartesian unification results of two spaces. A 3D geometry space is naturally good for a 3D variation space. In one scene, we can easily display five properties of an object: the Cartesian coordinates X, Y, Z, color, and size. If necessary, we can also extend to six properties by adding shape. Therefore, I have chosen to use objects in a 3D Cartesian coordinate system as nodes. When displayed on the screen, a variation space may be shown as a matrix of 3D points, which will allow us to easily see the organization of the space.

The users of my prototype were within a small group (composed primarily of the supervisors, myself, and another researcher, Zhenyu Qian, for investigator triangulation). To give users freedom to use this prototype by expressing diverse inputs and operations, I built an interactive panel accepting command lines and displayed results in a 3D window.
(Figure 8.1). The left side are two text panels. The user can input a block of commands in the lower panel. The top panel displays all executed commands and text results. 3D geometric results are shown in the right side 3D window. A series of input commands can be saved into a plain text file. The file can then be modified by text editor and reloaded into the prototype. Following is an example of the script:

```java
v3d = new View3d(); // create a new 3D window
vx = new vs("a","x",new int[] {10,20,30,40,50,60,70,80}); // initialize a space vx with 8 nodes with a.x property
vy = new vs("a","y",new int[] {10,20,30,40,50,60,70,80}); // space vy has 8 nodes with a.y
vz = new vs("a","z",new int[] {10,20,30,40,50,60,70,80}); // space vz has 8 nodes with a.z
v3d.show(vx.displayableClone("")); // display vx in the 3D window
v3d.show(vy.displayableClone(""));
v3d.show(vz.displayableClone(""));
v3d.addText("vx",85,5,0); // add the label "vx" in 3D
v3d.addText("vy",5,85,0);
v3d.addText("vz",5,10,85);
pause(); // pause the script
capture(v3d,"top","cartesian_input.tex", "vx","vy","vz"); // capture the 3D result into latex file. The viewing angle is top.
pause();
vxy = uy.unionUnify(vx,vy); // Union indexing unify vx and vy
v3d.show(vxy.displayableClone(1,10));
vvz = uy.unionUnify(vx,vz);
v3d.show(vxz.displayableClone(9,10));
capture(v3d,"front","index_cartesian_0.tex","vxy",);
pause();
vvxyz = uy.cartesian(vxy,vxz); // Cartesian unify vxy and vxz
v3d.show(vxyz);
pause();
capture(v3d,"front","index_cartesian_1.tex");
v3d.removeLast(); // remove last result (vxyz) from v3d
```
25 pause();
26 vxzy = uy.unionUnify(vxy, vxz);
27 v3d.show(vxyz);
28 capture(v3d,"front","index_cartesian_2.tex");
29 save("index_cartesian.txt"); // save above lines into a text
    file

The user can reload the saved script into the input panel and execute it block by block. Blocks are separated by the command line pause(). Each click on the Execute button runs the script till next pause(). The user can use such pauses to examine the intermediate result.

This prototype type is built with Java. The Java 3D API (Oracle, 2011) is used to display nodes in 3D space with different colors and sizes. BeanShell is used for scripting. The scripting syntax is very similar to Java (BeanShell, 2010).

8.2 Basic Objects and Methods

Following the formalism discussed in Chapter 6, I have implemented four types of objects: order type, node, variation head, and variation space. A variation space contains a list of variation heads. A variation head contains one or many nodes. A node uses order types as its values.

8.2.1 Order Types

Order types have been extensively studied by Chang (1999) and are not the focus of my research. Our interest in order types is to see the result of subsumption caused by order types in the space unification rather than in characters of order types. For demonstration purposes, I have implemented three order type objects: lifted integers, ascending integers, and descending integers.

Lifted Integers

As defined by Chang (1999), “Given the ordered set \( Z = (\mathbb{Z}, \bullet) \) where \( a \bullet b \) iff \( a = b \) we form \( \mathbb{Z}_\perp \), which we called the lifted integers by adding an element \( \perp, \mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\} \)" (Figure 8.2). An order relation “\( \bullet \)” is defined on \( \mathbb{Z}_\perp \) as \( \mathbb{Z}_\perp = (\mathbb{Z}_\perp, \bullet) \). For the order \( \bullet \) relation \( \bullet \) and \( a, b \in \mathbb{Z}_\perp \),
Figure 8.2: Order relation of lifted integers

- \( a \preceq b \) if \( a = b \)
- \( a \preceq b \) if \( a = \bot \) and \( b \in Z_{\bot} \)

For unification \( \sqcup \) and subsumption \( \sqsubseteq \), if \( a \in \overline{Z_{\bot}} \) and \( b \in \overline{Z_{\bot}} \),

- \( a \sqsubseteq b \) if \( a = b \) or \( a = \bot \)
- \( a \sqcup b = a \) (or \( b \)) if \( a = b \)
- \( a \sqcup b = \top \) if \( a \neq b \) and \( a \neq \bot \) and \( b \neq \bot \)

For example,

- \( \bot \sqsubseteq 2 \)
- \( 2 \sqsubseteq 2 \) (But not \( 2 \sqsubseteq 3 \))
- \( \bot \sqcup 2 = 2 \)
- \( 2 \sqcup 3 = \top \)

**Ascending Integers**

“Given the ordered set \( \mathbb{Z} = \langle \mathbb{R}, \leq \rangle \) where \( \leq \) denotes the usual “less than or equals” we form \( \overline{Z_{\bot}} \) which we call *ascending integers* by adding an element \( \bot \) to \( Z \), \( Z_{\bot \leq} = Z \cup \{ \bot \} \)” (Chang, 1999) (Figure 8.3). For the order relation \( \leq \) and \( a, b \in \overline{Z} \):

- \( a \leq b \) if \( a \leq b \in Z_{\bot \leq} \)
- \( a \leq b \) if \( a = \bot \) and \( b \in Z_{\bot} \)
For unification $\sqcup$ and subsumption $\sqsubseteq$, if $a \in \mathbb{Z}_{\perp \leq}$ and $b \in \mathbb{Z}_{\perp \leq}$, 

- $a \sqsubseteq b$ if $a \leq b$ or $a = \perp$
- $a \sqcup b = b$ if $a \leq b$
- $a \sqcup b = b$ if $a = \perp$
- $a \sqcup b = a$ if $b = \perp$. There is no $\top$

For example,

- $\perp \sqsubseteq 2$
- $2 \sqsubseteq 3$
- $\perp \sqcup 2 = 2$
- $2 \sqcup 3 = 3$

**Descending Integers**

“Given the ordered set $\mathbb{Z} = \langle \mathbb{Z}, \geq \rangle$ where $\geq$ denotes the usual “greater than or equals” relation, we form $\mathbb{Z}_{\perp \geq}$ which we call the **descending integers** by adding an element $\perp$ to $\mathbb{Z}$, $\mathbb{Z}_{\perp \geq} = \mathbb{Z} \cup \{ \perp \}$” (Chang, 1999) (Figure 8.4). For the order relation $\geq$ and $a, b \in \mathbb{Z}$:
Figure 8.4: Order of descending integers

- $a \geq b$ if $a \geq b \in \mathbb{Z}_\geq$
- $a \geq b$ if $a = \bot$ and $(b \in \mathbb{Z}_\bot)$

For unification $\sqcup$ and subsumption $\sqsubseteq$, if $a \in \mathbb{Z}_\geq$ and $b \in \mathbb{Z}_\geq$,

- $a \sqsubseteq b$ if $a \geq b$ or $a = \bot$
- $a \sqcup b = b$ if $a \geq b$
- $a \sqcup b = b$ if $a = \bot$
- $a \sqcup b = a$ if $b = \bot$. There is no $\top$

For example,

- $\bot \sqsubseteq 2$
- $3 \sqsubseteq 2$
- $\bot \sqcup 2 = 2$
- $2 \sqcup 3 = 2$
Construction Methods

The construction methods of the above three order type objects are:

1. lint(Integer x); // lifted integer
2. asc(Integer x); // ascending integer
3. desc(Integer x); // descending integer

For each type, the objects to represent an \(\bot\) are: lint.unbound, asc.unbound, and desc.unbound.

8.2.2 Nodes

To visually examine a node, variation head, and variation space, I have designed a node to display as a sphere in a 3D geometry space. Except for its name, a fully defined node has 4 properties: color and its locations in the 3D coordination system, X, Y, and Z. The values of these four properties can be any of the above order types.

A node can be constructed in the following ways:

1. node(String nodeName); // creates a node with only a name. All other properties are unbound
2. node(String nodeName, String propertyName, orderType value); // a node with only one property assigned with a value.
3. node(String nodeName, orderType X, orderType Y, orderType Z); // a node with X/Y/Z position. Color and size are unbound.
4. node(String nodeName, orderType X, orderType Y, orderType Z, orderType Color, orderType Size); // a fully defined node.

To display a partial node (a node with some properties unbound) in 3D space, we have to assign a value to all of the unbound properties. The method node.displayableClone() will create a new node based on the existing node. This new node inherits all assigned properties, and assigns default values to unbound properties. Default values of X, Y, and Z are 0 (zero). Default color is 0 (blue). For example, if a node has only X property assigned, it will shown up as a blue node on the X axis.
8.2.3 Variation Head

As defined in Chapter 6, a variation head contains one or several nodes where each has a unique name. In this prototype, a variation head is a Hashmap that uses a string (the node’s name) as the key with a value of the node. If a new node with the same name is added into the variation head, it will replace the old node of that name.

To construct a variation space:

\[
\text{vh\{node Node\}} \quad // \text{a head with one node}
\]
\[
\text{vh\{node... Nodes\}} \quad // \text{a head with several nodes in an array}
\]

To add a node into a variation head vHead:

\[
vHead.addNode(Node) \quad // \text{add a node Node into the variation head}
\]

8.2.4 Variation Space

A variation space is an indexed array, implemented as a Hashmap. Indices are tuples of integers. An index is denoted as a tuple of integers inside a square basket [...] (e.g., [0,1,0]). The length of the tuple is the dimension of the space. For each index there is a corresponding variation head.

The Index and the Index Set

An index is an tuple of integers. The method to construct an index object is:

\[
\text{index(int... Integers)};
\]

For example,

\[
\text{index d1a = new index(1);}
\]
\[
\text{index d2a = new index(1,1);}
\]
\[
\text{index d3b = new index(1,2,3);}
\]

All indices of a space form the index set of the space, such as \([0,0], [0,1], [1,0], [1,1]\), which is actually the index set of a 2×2 2-D space.
In my implementation, an index may have one or more *unbound* in its tuple. Its meaning depends on the script command that used it as an argument. In some cases (e.g., slice a space), this index represents a subset of an index set. Unbound means all the values along the dimension. \([\text{Unbound, 0}]\) means an index set of \([\{0,0\}, \{1,0\}]\). In other cases (e.g., select a boundary of a space), *unbound* means do not query on that dimension.

**Construction of a Variation Space**

To construct a new variation space:

\[
\text{vs vSpace = new vs();}
\]

To add a variation head into the space:

\[
\text{vSpace.add(Index, vHead) // assign the variation head vHead to the index Index.}
\]

A shortcut to assign many variation heads into a 1-D space:

\[
\text{vs vSpace = new vs(vh... vHeads); // assign a list of variation heads into the space.}
\]

Indices are by default incremental integers like \([0],[1],[2],[3],[4]...\).

**Displaying a Variation Head with 3 Nodes in the System**

Figure 8.5 shows a variation head and its 3 nodes in 3D space.

A node has 4 properties: colour and three locations \(x, y,\) and \(z\). The fuzzy boundary indicates that the node misses some location data (not all \(x, y,\) and \(z\) locations are bound with value). If the location value is unbound, to display it properly, the value is by default set to 0. For example, the fuzzy A node on the XY plane means that it is missing the Z value. For a node on the X axis with a fuzzy boundary, this normally means that it only has the X value bounded. But occasionally one of its X or Y values may be zero. A node’s colour is in the range from 0 (blue) to 100 (red). If a node has no colour assigned, by default its colour is 0 and it is displayed as blue.

A variation head is a collection of uniquely named nodes, i.e., in this example, \(\text{vhead = newvh(nodeA, nodeB, nodeC).}\)
Displaying a Variation Space in the System

A variation space is a collection of variation heads and will be displayed similarly to Figure 8.6. In most of these examples (including this one), a variation head only contains one node.
8.3 Operations on Indices

To better demonstrate the change of indices, in examples of this section, the nodes are placed as a result of their indexes. The node’s X/Y/Z position represents three dimensions: x for the first, y for the second, and z for the third dimension.

Operations demonstrated here include:

- Index shifting: shift indices of some variation heads
- Index distribution: loosen the space by evenly inserting empty slots along one or several dimensions.
- Append and Prepend index: convert a low-dimension space into high-dimension space by adding one dimension at the first (prepend) or the end (append) of current index
- Boundary: get the index boundary (maximum and minimum) of the space
- Slice: get a slice of the space

8.3.1 Index Mapping

The following examples demonstrate operations in the family of index mapping. Index mapping function changes variation heads’ position (indices) by converting selected indices into new indices through a user defined mapping process (e.g., a function). The 2-D space in Figure 8.7 is the original space that will be used in Figure 8.8, 8.9, and 8.10. I denote the integer tuple of the index as \([i_0, i_1]\).

![Indices of vsa](image)

**Figure 8.7:** The input of index mapping examples
Index Mapping - Index Shifting

Figure 8.8 shows two examples of shifting indices. Figure 8.8a moves the first index by 3 and the second index by 2 ([j₀, j₁] = [i₀ + 3, i₁ + 2]). In Figure 8.8b, if the node’s index 1 is greater than 3, move index 1 by 1; if the node’s index 2 is greater than 2, move index 2 by 2 (\([k₀, k₁] = [i₀ + 1, i₁ + 2]\) if \([i₀ > 3 \text{ and } i₁ > 2]\)).

Shift the indices of Figure 8.7 to make a new space vsb:

1. \(\text{idxShift} = \text{new indexShift(new index}(3, 2))\);
2. \(\text{vsb} = \text{uy.indexMapping(vsa, idxShift)}\);
   a: \([j₀, j₁] = [i₀ + 3, i₁ + 2]\)

Shift the indices of vsb to make a new space vsc:

1. \(\text{idxShift1} = \text{new indexShift(new index}(3, 2), \text{new index}(1, 2))\);
2. \(\text{vsc} = \text{uy.indexMapping(vsa, idxShift1)}\);
   b: \([k₀, k₁] = [i₀ + 1, i₁ + 2]\) if \([i₀ > 3 \text{ and } i₁ > 2]\)

Figure 8.8: Index shifting examples
Index Mapping - Index Distribution

In Figure 8.9, the new space’s variation head has a index of \([l_0, l_1] = [i_0 \times 2, i_1]\). This operation evenly inserts empty spaces along the first index. In Figure 8.9b, the new indices is \([m_0, m_1] = [i_0 \times 2, i_1 \times 2]\).

\[
\begin{align*}
\text{idxDistribute1} &= \text{new indexDistribute(\text{new index}(2,1))}; \\
\text{vsd} &= \text{uy:indexMapping(vsa, idxDistribute1)}; \\
c: [l_0, l_1] = [i_0 \times 2, i_1] \\
\text{idxDistribute1} &= \text{new indexDistribute(\text{new index}(2,2))}; \\
\text{vse} &= \text{uy:indexMapping(vsa, idxDistribute1)}; \\
d: [m_0, m_1] = [i_0 \times 2, i_1 \times 2]
\end{align*}
\]

\textbf{Figure 8.9:} Index distribution examples
Append or Prepend Index

Append and prepend add dimension(s) to a variation space. Append add a dimension(s) at the end of indices of \( \text{vsa} \) (Figure 8.10e). Prepend add a dimension(s) at the head of indices (Figure 8.10f).

```cpp
1. idxAppend = new indexAppend(new index(2));
2. vsf = uy.indexMapping(vsa, idxAppend);
   e: \([n_0, n_1, n_2] = [i_0, i_1, 2]\)

1. idxAppend = new indexPrepend(new index(2));
2. vsg = uy.indexMapping(vsa, idxAppend);
   f: \([0_0, 0_1, 0_2] = [2, i_0, i_1]\)
```

Figure 8.10: Append and prepend one dimension

### 8.3.2 Boundary of Space

Below examples show the boundary (maximum and minimum) of the index. Since the index could be arranged by the user intentionally, and each dimension may reflect to one property if the \( n - D \) space is constructed by Cartesian unification of \( n \) single property spaces, the boundary of index gives the highest (or lowest) priority variation heads along one or several properties. The script commands to select boundary are:
When used as an argument, an index here can have _unbound_ in its value tuple (e.g., [1, Unbound, Unbound]) to indicate no query on this dimension. Non-unbound value means query at this dimension.

**Boundary of a 2D space**

Figure 8.11 demonstrates the method to retrieve different boundaries of a 2D space vs0.

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>a: the input 2D space</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>b: the first heads at dimension 1</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>c: the last heads at dimension 2</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>d: the first and last heads at all dimensions</td>
</tr>
</tbody>
</table>

```python
a: vs1 = uy.selectMin(vs0, newindex(Unbound, 1))
b: vs1 = selectMax(vs0, new index(Unbound, 1))
c: vsTL = selectMax(vs0, new index(1, 1))
   vsBR = uy.selectMin(vs0, new index(1, 1))
```

**Figure 8.11:** Boundaries of a 2-D space
Boundary of a 3D space

Figure 8.12 shows several boundaries of a 3D space.

- **a:** The input 3D space vs0
- **b:** the last heads at dimension 3
- **c:** the last heads at dimension 1 and 2
- **d:** The last head at all dimensions

\[
\begin{align*}
\text{b: } & \text{vs1} = \text{selectMax}(\text{vs0, newindex(Unbound, Unbound, 1)}); \\
\text{c: } & \text{vs2} = \text{selectMax}(\text{vs0, newindex(1, 1, Unbound)}); \\
\text{d: } & \text{vs3} = \text{selectMax}(\text{vs0, newindex(1, 1, 1)});
\end{align*}
\]

**Figure 8.12:** A 3D space for selecting boundary
8.3.3 Slice

The slice operator gives a subspace of a given variation space (Figure 8.13). The subspace contains all variation heads with matching index in the old space. As an argument, an index can use Unbound in its tuple to indicate selecting all values in this dimension.

\[
\text{a: The original space} \quad \text{b: a slice at } Z \\
\text{c: a slice at } Y \quad \text{d: a slice at } Y \text{ and } Z \\
\text{e: a slice at } X \text{ and } Z \quad \text{f: a slice at } X, Y, \text{ and } Z
\]

\[
b: vxyz1 = vxyz.slice(new \text{ index(Unbound, Unbound,2)}); \\
c: vxyz2 = vxyz.slice(new \text{ index(Unbound, 2, Unbound)}); \\
d: vxyz3 = vxyz.slice(new \text{ index(Unbound, 2, 1)}); \\
e: vxyz4 = vxyz.slice(new \text{ index(1, Unbound, 1)}); \\
f: vxyz5 = vxyz.slice(new \text{ index(2,2,1)});
\]

**Figure 8.13:** Retrieve a subspace by slicing at given index
8.4 Indexing Unification

8.4.1 Six Types of Indexing Unifications

The Inputs

Inputs are the two variation spaces $vsx$ and $vsy$ in Figure 8.14.

$$vsx = \{[2]:a: [x:20, y: \bot, z: \bot], [3]:a: [x:30, y: \bot, z: \bot], [4]:a: [x:40, y: \bot, z: \bot], [5]:a: [x:50, y: \bot, z: \bot], [6]:a: [x:60, y: \bot, z: \bot], [7]:a: [x:70, y: \bot, z: \bot]\}$$

$$vsy = \{[4]:a: [x: \bot, y:40, z: \bot], [5]:a: [x: \bot, y:50, z: \bot], [6]:a: [x: \bot, y:60, z: \bot], [7]:a: [x: \bot, y:70, z: \bot], [8]:a: [x: \bot, y:80, z: \bot], [9]:a: [x: \bot, y:90, z: \bot]\}$$

Figure 8.14: Inputs of indexing unification
CHAPTER 8. SIMULATION OF THE FORMALISM

Union: \( v_{sx} y = v_{sx} \cup u v_{sy} \)

Union unification get a space with 8 variation heads (Figure 8.15).

Intersection: \( v_{sx} y = v_{sx} \cap l v_{sy} \)

Intersection unification get a space with 4 variation heads (Figure 8.16).

Figure 8.15: Union indexing unification

Figure 8.16: Intersection indexing unification
**Left Unification:** \( \text{vsxy} = \text{vsx} \uplus_L \text{vsy} \)

Left unification get a space with the same index as space \( \text{vsx} \) (Figure 8.17).

**Right Unification:** \( \text{vsxy} = \text{vsx} \uplus_R \text{vsy} \)

Right unification get a space with the same index as space \( \text{vsy} \) (Figure 8.18).
CHAPTER 8. SIMULATION OF THE FORMALISM

Left Difference Unification: \( vsxy = vsx \cup_{LD} vsy \)

Left difference unification get a space with two heads (Figure 8.19).

![Figure 8.19: Left difference unification](image)

Right Difference Unification: \( vsxy = vsx \cup_{RD} vsy \)

Left difference unification get a space with two heads (Figure 8.20).

![Figure 8.20: Right difference unification](image)
CHAPTER 8. SIMULATION OF THE FORMALISM

Cartesian Unification

As a contrast, Figure 8.21 is the Cartesian unification of vsx and vsy.

\[ \text{vsxy} = \text{vsx} \uplus \text{vsy}. \]

Figure 8.21: Cartesian unification result

8.4.2 Indexing Unification of Several 1-d Spaces

The three 1-D spaces \( vx, vy, \) and \( vz \) in Figure 8.22 are inputs for Figure 8.23.

Unify three pairs of spaces from Figure 8.22 by intersection unification will get three
1-D spaces in Figure 8.23a. Continuing the unification process will get the 1-D space in Figure 8.23b. Although variation heads in the result spaces are organized in the form of 2D or 3D grids, they are still 1-D spaces.

\[v_{xy} = v_x \cup_U v_y;\]
\[v_{xz} = v_x \cup_U v_z;\]
\[v_{yz} = v_y \cup_U v_z;\]

\[v_{xyz} = v_{xy} \cup_U v_z;\]
\[v_{xyz} = v_{xz} \cup_U v_y;\]
\[v_{xyz} = v_{yz} \cup_U v_x;\]
\[v_{xyz} = v_{xyz} \cup_U v_yz;\]
\[v_{xyz} = v_{xyz} \cup_U v_xz;\]
\[v_{xyz} = v_{xyz} \cup_U v_xz;\]
Three Ways of Combining Two Spaces by Index Mapping

Figure 8.24 demonstrates several ways of combining two spaces $vsa$ and $vsb$. In the figure, nodes’ positions reflect the variation heads’ indices. Figure 8.24b uses union unification to unifies variation heads with the same index, which put one space directly on top of the other space. Figure 8.24c and 8.24d uses index mapping to process indices first, then proceed with union unification.

**Figure 8.24:** Different ways of combining two spaces.
8.5 Cartesian Unification

8.5.1 Variation Head Cartesian With Variation Space

Figure 8.25 demonstrate how a single variation head ($vhx$, $vhz$, or $vhyz$) can be unified with a 2-d space ($vsxy$). Figure 8.25b assigns $Z$ property carried by $vhz$ to $vsxy$ to make positions ($X$, $Y$ and $Z$) of variation heads in $vsxyz$ fully defined. Due to subsumption, $vhx$ filters out many variation heads in $vsxy$ due to confliction of $X$ value in Figure 8.25c. Figure 8.25d assigns $Z$ property, but also filters out variation heads by $Y$ property.

![Cartesian unification of a variation head to a space](Figure 8.25)
8.5.2 1-d Space Cartesian Unifications

The inputs are three 1-D spaces $v_x$, $v_y$, and $v_z$ (Figure 8.26a). Unification of each pair gives us results in Figure 8.26b). Continuing the unification of spaces in Figure 8.26a and Figure 8.26b gives a 3-D space (Figure 8.26c).

**Figure 8.26**: Cartesian unification of 1-D spaces
8.5.3 Several Examples of Cartesian Unifications

Example I

Continue to use the three 1-D spaces ($vx$, $vy$, and $vz$) from above section, first creating three more 1-D spaces by union indexing unification (Figure 8.27). Cartesian unification of each pairs gives us results in Figure 8.28.

\[
\begin{align*}
\text{vxy} &= vx \cup_U vy; \\
\text{vxz} &= vx \cup_U vz; \\
\text{vyz} &= vy \cup_U vz;
\end{align*}
\]

Figure 8.27: Cartesian unification example I - inputs
Figure 8.28: Cartesian unification example I - results
Example II

Unification of each pair in the three 1-D spaces of Figure 8.29a gets the same result as Figure 8.29b.

\[ vxyz4 = vxy \sqcup vyz \]
\[ vxyz4 = vxz \sqcup vyx \]
\[ vxyz4 = vxy \sqcup vyx \]

**Figure 8.29**: Cartesian unification example II
Example III

In Figure 8.30a, $vxy$ is a 1-d space, $vxyz2$ is a 2-d space (the same $vxyz2$ of Figure 8.28). Figure 8.30b is the result of $vxyz6 = vxy \sqcup vxyz2$. Cartesian unify other spaces from Figure 8.27 and 8.28 also gets the same result.

![Diagram of Cartesian unification example III](image)

$\begin{align*}
  vxyz6 &= vxy \sqcup vxyz2 \\
  vxyz6 &= vxz \sqcup vxyz1 \\
  vxyz6 &= vxy \sqcup vxyz1 \\
  vxyz6 &= vyz \sqcup vxyz2 \\
  vxyz6 &= vxy \sqcup vxyz2 \\
  vxyz6 &= vyz \sqcup vxyz3 \\
  vxyz6 &= vxz \sqcup vxyz3
\end{align*}$

**Figure 8.30:** Cartesian unification example III
Example IV

Take two spaces (Figure 8.31a) generated from Figure 8.28b and 8.28c, The unification $(vxyz5 = vxyz2 \cup \otimes vxyz3)$ gives us the result in Figure 8.31b. Unification of any pair of spaces from Figure 8.28 gives us the same result.

---

**Figure 8.31**: Cartesian unification example IV
Example V

\( vs1d \) is a 1-D space with three properties \((x, y, \text{ and colour})\) in each node. \( vs2d \) is a 2-D space with two properties \((x \text{ and } z)\) in each node (Figure 8.32a). The Cartesian unification result \( vs3d = vs1d \bigcup_{\otimes} vs2d \) is in Figure 8.32b.

\[ \begin{align*}
\text{Figure 8.32: Cartesian unification example V}
\end{align*} \]
Example VI

This example (Figure 8.33) demonstrates both Cartesian Unification and Indexing Unification. $vs_{xy}$ and $vs_{xz}$ are both 2-D variation spaces.

**Figure 8.33:** Cartesian unifications vs. indexing unifications
8.6 Mixed Unification

While working on the above Cartesian unification examples, we see one problem, where the dimension of the space gets large and the number of variation heads in the space becomes huge. Sometimes we do not need the full Cartesian unification. We may rather want to mix the indexing unification with the Cartesian unification to employ a mixed method. In this method, the first step is to use indexing unification along one (or several) dimension, then to make the Cartesian unification for the remaining dimensions. A m-D space $\bigcup_a$ a n-D space will lead to a m+n dimensions space. But as a mixed method, if the indexing unification involves $P$ dimensions, then the final space will have $m + n - p$ dimensions. For example, in Section A, the original inputs are three 1-D spaces: $vx$, $vy$, and $vz$, with the lengths of the 5, 6, and 4 variation heads in each of them. Through Cartesian unification, we get three 2-D spaces, $vxz$, $vxy$, and $vyz$, with the lengths of the 20, 30, and 24 variation heads. But if we continue to unify these 2D spaces (for example, $vxzyz = vxz \bigcup_v yz$), the space $vxzyz$ will be a 4-D space, with $20 \times 24 = 480$ variation heads. In here, actually $vxzyz = vx \bigcup_v y \bigcup_v x \bigcup_v vz$. For order type lifted number ($a \sqsubseteq b$ if and only if $a = \bot$ or $a == b$), $vx \bigcup_v x$ will have the same number of valid (non-top) variation heads as $vx$. Therefore, $vxyxz$ will have exactly the same number of unique variation heads of $vxyz = vx \bigcup_v y \bigcup_v vz = vxy \bigcup_v vz$. For such a case, we may want to use the mixed method to index unify the first dimension (x values) and then Cartesian unify the other two dimensions (y and z values).

The indexing unification method involved in this mixed method could be one of the six indexing unification methods. Below I just use the intersection unification to describe the procedure for the mixed unification of $V_a$ and $V_b$ into $W$ along the dimension $d$ (this could be more than one dimensions $\{d_0, \ldots, d_n\}$):

1. Get the index set $I_a$ that contains all index values of $V_a$ along dimension $d$.
2. Get the index set $I_b$ that contains all index values of $V_b$ along dimension $d$.
3. $I = I_a \cup I_b$ // Depends on the indexing unification method, get the union/intersection/difference of $I_a$ and $I_b$.
4. For each index value $i$ in $I$:
   $W_a = slice(V_a, i)$
\textit{W}_b = \text{\textit{slice}}(V_{-b}, i) \quad // \text{Slice } V_a \text{ and } V_b \text{ using } i.
\\
\textit{W}_i = V_a \bigcup_\otimes V_b \quad // \text{Cartesian unify } W_a \text{ and } W_b:
\\
\text{For each variation head } v_w \text{ and its index } i_w \text{ in } W_i
\\
\quad \quad j = [i, i_w] \quad // \text{Create a new index } j \text{ by prefix } i \text{ to } i_w
\\
\quad \quad W[j] = v_w \quad // \text{Assign } v_w \text{ into } W

If the participant multi-dimensional spaces are generated from several 1-D spaces, then the mixed method is equalized to index unify the required 1-D spaces, and then to Cartesian unify with remaining 1-D spaces. For example, with the 4 1-D spaces \(v_1, v_2, v_3,\) and \(v_4, v_{12} = v_1 \bigcup_\circ \bigcup_\otimes v_2, v_{34} = v_3 \bigcup_\otimes v_4,\) then \(v_{12}\) mixed with \(v_{34}\) through indexing unification of the first dimension is \(v_{1234} = (v_1 \bigcup_\otimes v_2) \bigcup_\otimes v_3 \bigcup_\otimes v_4.\) If the indexing unification is on the 2nd dimension, then \(v_{1234} = v_1 \bigcup_\otimes v_2 \bigcup_\otimes (v_3 \bigcup_\otimes v_4).\)

In the script, the mix operation can be done by one of the following:

1. \texttt{variationSpace mix(variationSpace vs1, variationSpace vs2, int i);}
2. \texttt{variationSpace mix(variationSpace vs1, variationSpace vs2, index idx);}  

In the above first command, it matches the first dimension of space \(vs1\) to \(\{i + 1\}\)th dimension of space \(vs2.\) In the second command, the index \(idx\) argument indicates which dimension(s) are used to match. For example, if both \(vs1\) and \(vs2\) are 3-D spaces, then \(idx\) should be a 3-tuple index. In the tuple, \textbf{unbound} indicates not to match in this dimension. Non-unbound numbers indicate to match at this dimension. For example, \([1, \text{unbound}, \text{unbound}]\) means to match at the first dimension and leave the other two dimensions in Cartesian unification; \([\text{unbound}, 1, 1]\) means to match at the second and third dimension but leave the first dimension in Cartesian unification.

### 8.6.1 Apply Colour Onto One Dimension

Figure 8.34 demonstrates mixing one 1-D space with a 3-D space. There are two spaces as inputs: One is a 1D space \textit{colour} with only colour values. The second is a 3-D space (Figure 8.34a). The content of \textit{colour} space is:

\texttt{colour = \{[0]:\{a:[c:0]\}}, [1]:\{a:[c:10]\}}\texttt{, [2]:\{a:[c:20]\}}, [3]:\{a:[c:30]\}, [4]:\{a:[c:40]\}, [5]:\{a:[c:50]\},  
[6]:\{a:[c:60]\}, [7]:\{a:[c:70]\}, [8]:\{a:[c:80]\}, [9]:\{a:[c:90]\}}\texttt{]}

 CHAPTER 8. SIMULATION OF THE FORMALISM

Figure 8.34: Mixing example - applying colour on one dimension

**Figure 8.34**

- **a:** The original 3D space
- **b:** mix on the first dimension
  
  \[
  v_{c1} = \text{mix}(\text{color}, \text{vxyz}, 0);
  \]

- **c:** mix on the second dimension
  
  \[
  v_{c2} = \text{mix}(\text{color}, \text{vxyz}, 1);
  \]

- **d:** mix on the third dimension
  
  \[
  v_{c3} = \text{mix}(\text{color}, \text{vxyz}, 2);
  \]
8.6.2 Mix a 1-D Space With a 2-D Space

Figure 8.35 demonstrates mixing a 1-D space ($vxy$) with a 2-D space ($vxz$). $vxy$ has variation heads with only $X$ and $Y$ values. $vxz$ has variation heads with only $X$ and $Z$ values. Due to subsumption and unification, the mixed result in Figure 8.35 only has 6 variation heads.

Figure 8.35: Mix a 1-D space with a 2-D space
8.6.3 Mix Two 2-d Spaces

Figure 8.36 demonstrates mix unification of two 2-D spaces $vxy$ and $vxz$. With regular Cartesian unification, $vxy \sqcup_vxz$ will result in a 4-D space. With mix unification, depending on the option, the result could be a 3-D space or 2-D space. Figure 8.36 mixes two spaces on two dimensions, which is actually the intersection indexing unification.

---

**Figure 8.36**: Mix two 2-D spaces
8.7 Order Type Unifications

Order type unification makes the result semantically rich. In examples in this section, nodes’ values are reflected by their positions. There may exist several variation heads within the same location.

8.7.1 Indexing Unification

Figure 8.37 shows the effect of ascending and descending integers in indexing unification. The inputs are two 1-D spaces $vsx_0y$ and $vsx_1y$. Indices of variation heads for both spaces start from bottom left to top right corner. The result is $vsx_2y = vsx_0y \sqcup_I vsx_1y$.

Both X and Y are Ascending Integers (Figure 8.37a)

- $a \sqsubseteq b$ if $a \leq b$ or $a = \perp$
- $a \sqcup b = b$ if $a \leq b$ or $a = \perp$

For a node with both X and Y are ascending integers, we can read it as “if X’s value is at least $a$, Y’s value is at least $b$, and vise versa.”.

Both X and Y are Descending Numbers (Figure 8.37b)

- $a \sqsubseteq b$ if $a \geq b$ or $a = \perp$
- $a \sqcup b = b$ if $a \geq b$ or $a = \perp$

For a node with both X and Y are descending integers, we can read it as “if X’s value is at most $a$, Y’s value is at most $b$”.

X is an Ascending Number and Y is a Descending Number

For a node with X as ascending and Y as descending integers, we can read it as “if X’s value is at least $a$, Y’s value is at most $b$”.

X is a Descending Number and Y is An Ascending Number

For a node with X as descending and Y as ascending integers, we can read it as “if X’s value is at most $a$, Y’s value is at most $b$”.
a: the input

b: both X and Y are ascending integers
c: both X and Y are descending integers
d: X is ascending, Y is descending
e: X is descending, Y is ascending

Figure 8.37: Indexing unification of heads with ascending or descending integers
8.7.2 Cartesian Unification

One Head to a 2-D space

$vhx$ is variation heads. $vsxy$ is a 2-D space. These nodes’ $X$ property carries ascending integer, $Y$ carries descending integers. With Cartesian unification, the result space $vsxyx$ has the same dimension as $vsxy$. Due to subsumption, $X$ of nodes in $vsxyx$ are greater or equal than $X$ of $vhx$. (Figure 8.38b). Similarly, in Figure 8.38c, the result $vsxyz$’s $Y$ values are less or equal than $Y$ of $vhyz$.

![Diagram](image)

Figure 8.38: Unify two variation heads to a 2-D space with order types
CHAPTER 8. SIMULATION OF THE FORMALISM

Order Type Cartesian Unification

In this example, I uses two 1-D spaces in the X/Y plane (Figure 8.39). Cartesian unification result can be seen in Figure 8.40.

Figure 8.39: Inputs of 1-D order type Cartesian unification

In Figure 8.40a, nodes’ X and Y are all ascending values, which means, when two variation heads unifies, the result head’s node keeps the bigger X and Y from the nodes of inputting heads. The unified node always located at the top right corner of the two unifying nodes. The node at the bottom left corner is the unification of two nodes at the bottom left corner of two spaces, which takes the larger X and Y values. The top right corner nodes is the unification of two nodes at the top right corner, which also takes the larger X and Y values from inputting nodes. The left boundary is defined by the larger X of two smallest X nodes, and the right boundary is defined by the larger X of two largest X nodes. The top boundary is defined by the larger Y of the two nodes with the largest Y, and the bottom boundary is defined by the larger Y of the two nodes with smallest Y.

In Figure 8.40b, nodes’ X is descending and Y is ascending values. When two variation heads unifies, the result head’s node keeps the smaller X but the bigger Y from the nodes of inputting heads. The unified node is always located at the upper left side of the two unifying nodes. Therefore, the new space’s nodes filled in the area of the upper left section with the upper left boundary of the two input 1-D spaces. The left boundary is defined by the smallest X of nodes in both spaces. The top boundary is defined by the largest Y of nodes.

In Figure 8.40c, nodes’ X is ascending and Y is descending value. When two variation heads unify, the result head’s node keeps the bigger X but the smaller Y from the nodes
of inputting heads. The unified node always located at the lower right side of the two unifying nodes. Therefore, the new space’s nodes filled in the area of the lower right section with the lower right boundary of the two input 1-D spaces. The right boundary is defined by the largest $X$ of nodes in both spaces. The bottom boundary is defined by the smallest $Y$ of nodes.

In Figure 8.40d, nodes’ $X$ and $Y$ are both descending values. When two variation heads unifies, the result head’s node keeps the smaller $X$ and the smaller $Y$ from the nodes of inputting heads. The unified node always located at the lower left side of the two unifying nodes. The right boundary is defined by the largest $X$ of nodes in both spaces. The bottom boundary is defined by the smallest $Y$ of nodes. Similar to Figure 8.40a, the boundaries are defined by the nodes with largest and smallest $X$ and $Y$ values, except the result nodes take the smaller $X$ and $Y$ from unifying nodes.

Figure 8.40: Cartesian unification of two 1-D spaces with order types
Another Example of Order Type Cartesian Unification

Figure 8.41 shows the Cartesian unification of two 1-D spaces with ascending or descending integers as their nodes’ value. Figure 8.41a and b’s three graphs show the same result but from three different view points.

**Figure 8.41**: Another example of order type Cartesian unification
8.7.3 Cartesian Square

Figure 8.42 shows the Cartesian square of 1-D spaces with order types. Due to subsumption, when \( X \) and \( Y \) carry different order types, the result nodes are located within four sections (upper right, upper left, lower right, and lower left) of the \( X/Y \) plane.

\[
\begin{align*}
\text{a: the input} & \\
\text{b: } X \text{ ascending, } Y \text{ ascending} & \\
\text{c: } X \text{ descending, } Y \text{ ascending} & \\
\text{d: } X \text{ ascending, } Y \text{ descending} & \\
\text{e: } X \text{ descending, } Y \text{ descending} & \\
\end{align*}
\]

\[
vssel = vsxy \sqcup vsxy
\]

**Figure 8.42:** Order type Cartesian square unification
8.8 Relational Algebra

In this section are examples of three of the primary operations (selection, rename, and projection) of relational algebra.

8.8.1 Select

Figure 8.43 uses select to create a new space with variation heads satisfying the selection criteria.

```java
vsxy1 = select(vsxy, new compare("a", "x", ",", "a", "y");
```

b: Select Variation Heads With \( a.x > a.y \)

```java
vsxy1 = select(vsxy, new compare("a", "x", ",", "a", "y");
```

c: Select Variation Heads With \( a.x \preceq a.y \)

```java
vsxy1 = select(vsxy, new compare("a", "x", ",", "a", "y");
```

d: Select Variation Heads with \( a.x = 30 \)

**Figure 8.43**: Select variation heads from a space
8.8.2 Rename

Figure 8.44 renames nodes or properties’ name. Each variation head in the input has two nodes \( a \) and \( b \). Nodes \( a \) are blue and \( b \) are red. Figure 8.44a shows the same graph as the original space because nodes’ name are not showing in the graph. But renaming properties’ name causes nodes to relocate (Figure 8.44b).

\[
\begin{align*}
\text{vsxy1} & = \text{rename(vsxy, new renameNode("a","c"))}; \\
\text{vsxy1} & = \text{rename(vsxy1a, new renameNode("b","d"))};
\end{align*}
\]

b: Rename nodes’ name \( a \) to \( c \), \( b \) to \( d \)

\[
\begin{align*}
\text{vsxy1a} & = \text{rename(vsxy, new exchangeNodeProperty("a","x","y"))}; \\
\text{vsxy1} & = \text{rename(vsxy1a, new exchangeNodeProperty("b","x","z"))};
\end{align*}
\]

c: Rename properties’s name \( a.x \) to \( a.y \), \( b.x \) to \( b.z \)

**Figure 8.44:** Rename name of nodes and properties
8.8.3 Projection

Each variation head in the input of Figure 8.45 has two nodes $a$ and $b$. Nodes $a$ are blue and $b$ are red. With projection, only projected nodes and properties are kept in the result (Figure 8.45a and b).

\[ X \quad Y \quad Z \]

\[ \text{vsxyz} \]

\[ a: \text{the input} \]

\[ X \quad Y \quad Z \]

\[ \text{vs1} \]

\[ b \]

\[ b: \text{project colour, } a.x, \text{ and } b.y \]

\[ X \quad Y \quad Z \]

\[ \text{vs1} \]

\[ c \]

\[ c: \text{project } a.x, a.z, b.x, \text{ and } b.y \]

\[ \text{vs1} = \text{project(vsxxyz, new projection("a.x","b.y","a.c","b.c"))}; \]

\[ b: \text{project colour, } a.x, \text{ and } b.y \]

\[ \text{vs1} = \text{project(vsxxyz, new projection("a.x","a.z","b.x","b.y"))}; \]

\[ c: \text{project } a.x, a.z, b.x, \text{ and } b.y \]

Figure 8.45: Project a variation space
8.9 Summary

In the above, I have used 3-D geometric data to simulate the formalism so that we can visually examine the input and the output. In this simulation system, a node can have four properties, X, Y, Z, and colour, and the values of the properties could be order types, such as ascending and descending numbers. Based on the algorithms proposed in the formalism, I have implemented index operations, indexing unification, Cartesian unification, and operations from relational algebra.

While working with this simulation system, we have found several useful operations which were not included in the designed formalism: for example, the slice to obtain a sub-space, and the mix unification method to combine indexing unification and Cartesian unification. I believe there may be many useful operators still waiting to be discovered.

Order type values bring many more varieties to the unification results. Although I only demonstrate the ascending and descending numbers within the order type section, we have seen that these results appear to be richer than the default lifted integers. If we had tried to simulate all order types, as suggested by Chang (1999), we may have ended up with a chapter which was ten times longer than the text included here.
Chapter 9

CZSaw Use Cases

In this chapter, I use different data sets to demonstrate how our formalism can help the user to handle alternatives in CZSaw. Since 2006, the IEEE VAST Challenge contests have provided visual analytics researchers with opportunities to evaluate and modify their tools through the use of benchmark data sets and realistic tasks. Analytical tasks and data sets are derived from these VAST contests.

In the current version of CZSaw, the dependency graph is a special case of the parametric model; all nodes are single property nodes. The value of a node could be an atomic value (number, string, etc.) or a CZSaw object (e.g., an individual or set of entities, an individual or set of relations, or a data view).

Applying one variation head $v$ to the symbolic model will drive CZSaw to a certain state, which we will call system result $r$. Batches that apply many variation heads in a variation space $V$ on the symbolic model will drive CZSaw to create a space of results $R$. $R$ has the same dimension and size as $V$. Each result $r$ in $R$ has its own index $i$, which is associated to a variation head $v$ that has the same index $i$ in $V$. The following sections describe several possible ways of generating variation spaces for different analytical tasks. Also, the analyst can interact with the generated results to study the problem further.
# Chapter 9. CZSAW Use Cases

## 9.1 Use Case A: VAST 2008 Mini Challenge 3 - Cell Phone Calls

### 9.1.1 Symbolic Model of the Solution

This challenge asked us to identify a social network structure and its change based on phone call records (section 2.2.2). One major problem of this challenge is to find replacement phones of several phones. We assume that the substitutes should have similar contacts (unless most contacts have also changed their numbers) and similar locations (towers) as had the original phone. Base on this assumption, Figure 9.1 shows an abstract model of the solution. The dependency graph created by CZSaw is much more complex, but the core logic and inputs are similar. This model provides us with more parameters than the one in Figure 2.7 of Section 2.2.2. Figure 2.7 only checks for similar contacts. This symbolic model allows us to check for similar contacts, similar locations, and for the confidence level of the replacement. All nodes in this model are single property nodes. There are three source nodes: `originalPhone`, `objectType`, and `relationThreshold`. The value of `originalPhone` is a phone number (or a set of phone numbers). The value of `objectType` is an object type such as “phone”, “tower”, or “date”. The `relationThreshold` is an integer. The successor node `relatedObjects` has a value of the constraint expression:

\[
\text{relatedObjects} = \text{relatedNodes}(\text{objectType}, \text{originalPhone})
\]

The evaluated result is a list of related objects with the type `objectType`. The `replacingPhone`’s value is:

\[
\text{replacePhone} = \text{commonRelated}(	ext{relatedObjects}, \text{"phone"}, \text{relationThreshold})
\]

Its evaluated value is a set of phones that are related to at least number of `relationThreshold` objects in `RelatedObjects`.

We need to check phone IDs 1, 2, 3, 5, 97, and 137. The variation space of `originalPhone` is \{1,2,3,5,97,137\}. The `objectType`’s space is \{“phone“, “tower”\}. The “phone” determines the value of the `relatedObjects` as list of phones talking to the original phone. The “tower” means that the evaluated value of `relatedObjects` are the actual towers to which the original phone connects (this can be used to indicate the active location of the phone). The `relationThreshold`’s value is an integer, which indicates that at least this number of `RelatedObjects` have a relationship with the `replacePhone`. This number measures the strength of similarity. The higher the number, the higher the similarity between the `replacePhone` and
Figure 9.1: The parametric model to find the replacement phone

`originalPhone`. We then define the space of the `relationThreshold` as \{1,2,3,-2,-1\}. 1 will give all phones that are related to the `Related Objects`. 2 and 3 get those phones that are related to at least 2 or 3 objects in `relatedObjects`. -2 give phones that connect to at least half of the objects, and -1 give phones that connect to all objects in `relatedObjects`. The key CZSaw commands involved are:

```plaintext
: relatedObjects = relatedNodes(objectType, originalPhone);
: replacePhone = commonRelated(relatedObjects, "phone",
                               relationThreshold);
```

Figures 9.2a and 9.2b demonstrate the resulting node-link graph of two variation heads. Figure 9.2a shows that for the given phone 2, there are several phones (e.g., phones 1, 3, 5, 397) that connect to at least three phones that talk to phone 2. But phone 397 has the most connections among these phones. Figure 9.2b shows that phone 309 connects to exactly all towers to which phone 1 connects, and this means that these two phones were active in the same region. These graphs suggest that we should further investigate the relationship of phones 2 and 397, and the relationship of phones 1 and 309. From other evidence (e.g., the calling time of the phones where phone 2 stopped at one day and phone 397 immediately started to participate in the network thereafter), we can conclude
CHAPTER 9. CZSAW USE CASES

a: {originalPhone:2, relationThreshold:3, objectType:phone}

b: {originalPhone:1, relationThreshold:-1, objectType:tower}

Figure 9.2: Two states of using CZSaw to find replacement phones
that phone 397 replaces phone 2, and that phone 309 replaces phone 1.

In this example, I use only the simple form of the subsumption from the lifted integers: $A \sqsubseteq B$ if $A$ is equal to $B$ or $A$ is $\bot$.

The Cartesian unification $\text{originalPhone} \sqcup \text{objectType} \sqcup \text{relationThreshold}$ gives us the space of all possible combinations of the three source nodes (Table 9.2). Applying these variation heads from the space onto the parametric model (assigning the values to the source nodes) will drive CZSaw to compute and display different graphs in its graph view window (Table 9.1). Screenshots in this table are generated through CZSaw script and the dependency graph. CZSaw can load and run an external script. I wrote the script to loop through all variation heads in the variation space in sequential. With each variation head, the dependency graph update CZSaw to generate a corresponding screen. The script also automatically take a screenshot after each update. Then I manually organize these screenshots into the table.

In Table 9.1, each screenshot reflects the results from one variation head. Since a variation head here contains three nodes, to display these screenshots with the highest efficiency, I shall use a tabular format where screenshots in each column have the node $\text{relationThreshold}$ (table heading) with the same value, and the screenshots in each row have the same values on the two nodes $\text{originalPhone}$ and $\text{objectType}$ (under each row of screenshots). From these results, we can easily see the affects of the values of nodes $\text{objectType}$ and $\text{relationThreshold}$. The higher value of assigned to $\text{relationThreshold}$, the more restricted the value we can obtain for $\text{replacingPhone}$. When the $\text{relationThreshold}$ is a small number (e.g., 1 or 2), the graph is busy with many nodes as $\text{replacingPhone}$ (and edges among $\text{replacingPhone}$ and $\text{RelatedObjects}$). But when the number goes up to -1 ($\text{replacingPhone}$ has to connect to every entity in $\text{RelatedObject}$), frequently the value of $\text{replacingPhone}$ is equal to $\text{originalPhone}$ (where the only phone which connects to all of its related objects is itself). Since a tower can connect to many more phones than can a single phone, with the same $\text{relationThreshold}$, the number of phones found in the $\text{replacingPhone}$ is (almost) always more for the $\text{objectType}$ with the tower than with the phone.
<table>
<thead>
<tr>
<th>RelThres:1</th>
<th>RelThres:2</th>
<th>RelThres:3</th>
<th>RelThres:-2</th>
<th>RelThres:-1</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Image" /></td>
<td><img src="image2" alt="Image" /></td>
<td><img src="image3" alt="Image" /></td>
<td><img src="image4" alt="Image" /></td>
<td><img src="image5" alt="Image" /></td>
</tr>
<tr>
<td>originalPhone:1, objectType:phone</td>
<td>originalPhone:2, objectType:phone</td>
<td>originalPhone:3, objectType:phone</td>
<td>originalPhone:5, objectType:phone</td>
<td>originalPhone:97, objectType:phone</td>
</tr>
<tr>
<td><img src="image6" alt="Image" /></td>
<td><img src="image7" alt="Image" /></td>
<td><img src="image8" alt="Image" /></td>
<td><img src="image9" alt="Image" /></td>
<td><img src="image10" alt="Image" /></td>
</tr>
<tr>
<td>originalPhone:137, objectType:phone</td>
<td>originalPhone:1, objectType:tower</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 9.1 continued on next page**
Table 9.1 – continued from previous page

<table>
<thead>
<tr>
<th>RelThres:1</th>
<th>RelThres:2</th>
<th>RelThres:3</th>
<th>RelThres:-2</th>
<th>RelThres:-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Image](originalPhone:2, objectType:tower)</td>
<td>![Image](originalPhone:2, objectType:tower)</td>
<td>![Image](originalPhone:2, objectType:tower)</td>
<td>![Image](originalPhone:2, objectType:tower)</td>
<td>![Image](originalPhone:2, objectType:tower)</td>
</tr>
<tr>
<td>![Image](originalPhone:3, objectType:tower)</td>
<td>![Image](originalPhone:3, objectType:tower)</td>
<td>![Image](originalPhone:3, objectType:tower)</td>
<td>![Image](originalPhone:3, objectType:tower)</td>
<td>![Image](originalPhone:3, objectType:tower)</td>
</tr>
<tr>
<td>![Image](originalPhone:5, objectType:tower)</td>
<td>![Image](originalPhone:5, objectType:tower)</td>
<td>![Image](originalPhone:5, objectType:tower)</td>
<td>![Image](originalPhone:5, objectType:tower)</td>
<td>![Image](originalPhone:5, objectType:tower)</td>
</tr>
<tr>
<td>![Image](originalPhone:97, objectType:tower)</td>
<td>![Image](originalPhone:97, objectType:tower)</td>
<td>![Image](originalPhone:97, objectType:tower)</td>
<td>![Image](originalPhone:97, objectType:tower)</td>
<td>![Image](originalPhone:97, objectType:tower)</td>
</tr>
<tr>
<td>![Image](originalPhone:137, objectType:tower)</td>
<td>![Image](originalPhone:137, objectType:tower)</td>
<td>![Image](originalPhone:137, objectType:tower)</td>
<td>![Image](originalPhone:137, objectType:tower)</td>
<td>![Image](originalPhone:137, objectType:tower)</td>
</tr>
</tbody>
</table>

Table 9.1: Screenshots generated by CZSaw from the cellphone variation space

9.1.2 Possible Interactions

There are many ways to display these multiple results through a graphical user interface (GUI). Researchers have proposed many solutions, such as either spreadsheet-like views which put screenshot thumbnails onto a table or the parallel pie (Terry, 2005) which places multiple thumbnails into slices in a square block. A table view is, by nature, good for thumbnails that are organized as a 2-D array, like our 2-D variation heads. The horizontal axis lines up one (or several) properties, and the vertical axis lines up other properties. If we desire to fit multiple images that are organized as a high-dimensional array, rather than displaying such images in a 3-D window on a 2-D monitor, we could also choose to
Table 9.2: The result of Cartesian unification. Tuples are \{\text{originalPhone, objectType, relationThreshold}\}

employ Torgerson’s classical multidimensional scaling (MDS) method (Torgerson, 1958) in order to place the high dimensional organized images into a 2-D window, such as Design Galleries (Marks et al., 1997). However, this Ph.D research is not focused on the issue of displaying the layout of multiple results, but is rather intended to provide a formalism by which to guide both the layout and interaction designs of a system in order to support multiple results.

Once many results have been displayed in the interface (e.g., thumbnails in a spreadsheet format), the user can then interact with these multiple results in two ways, either by working with variation heads or by working with the final results (e.g., visually from the GUI).

Within any layout in a GUI that supports interactions, the user should be able to select one or several results at once (e.g., by mouse clicks). Based on such a selection, the system is able to retrieve the corresponding variation head(s). Then our formalism is able to support the system design, which allows the user to narrow down or navigate through the large number results based on these variation heads. To demonstrate such direct manipulation over these results, I will use a simple layout - a tabular view to display multiple system results as thumbnails in table cells, such as is shown in Table 9.1.
Interactions with Variation Heads

To work directly with variation heads, we can use the selection method to only select interesting variation heads and the results for us to study more carefully. We can first all possible result of one phone, e.g., phone 1. Then this operation becomes checking all variation heads that subsumed by:

\{originalPhone:1\}

The results are illustrated in Table 9.3.

To obtain results that contain more accurate and informative data about the replacement phones (i.e., which connect to the most similar phones and are active with the same towers), we can use a selection criteria such as: Any variation head that subsumed by:

\{objectType:tower, relationThreshold:-1\} or
\{objectType:phone, relationThreshold:-2\}

The results can be seen in Table 9.4.

We can also do the selection operation from the GUI, in which we are picking up one thumbnail (e.g., the fourth thumbnail at the first row). We are then able to retrieve its variation head

\{originalPhone: 1, objectType: phone, relationThreshold:-2\}

Then it becomes simple to obtain results such as those in Table 9.3 or in part of Table 9.4 by applying selection over the variation head.

Such a selection requires the user to switch between two modes: the abstract mode with variation heads, and the visual mode of thumbnails. The user has to first visually work with thumbnails in the GUI, check variation heads associated with these thumbnails, manipulate these variation heads, and finally update the GUI with a new set of thumbnails.
Table 9.3: Select variation heads that subsumed by {originalPhone=1}
Table 9.4 - Continued on next page
Table 9.4 – continued from previous page

![Diagram of two nodes: {phone, 137, -2} and {tower, 137, -1}]

Table 9.4: Select variation heads that subsumed by \{objectType: tower, relationThreshold: -1\} or \{objectType: phone, relationThreshold: -2\}

Direct Manipulation - Closely Related Results

Due to the limitations of the computer monitor (planar 2D), it is always challenging to properly display high-dimension results. In our examples of tabular layout, two results that have similar parameters may not be closely located (and many other layout algorithms have the same problem). For example, for two variation heads

\{originalPhone: 2, objectType: phone, relationThreshold: -2\} and
\{originalPhone: 2, objectType: tower, relationThreshold: -2\}

the only difference is for the node \text{objectType}, but in the Table 9.1, these two results are shown on row 2 and row 8, respectively. Sometimes we may want to access closely related results, such as for example, where the inputs only vary at one parameter, which is, in our variation heads, where a variation head only differs from one node to the other head and the difference of the node value is only by one index.

To demonstrate this with an example, the user can choose the thumbnail at row 2/column 4 in Table 9.1. Then the system shall retrieve the associated variation head

\{originalPhone: 2, objectType: phone, relationThreshold: -2\}

which has an index of \([1,0,3]\). Then the system looks up all closely-related variation heads, which have indexes of: \([0,0,3]\), \([2,0,3]\), \([1,1,3]\), \([1,0,2]\), and \([1,0,4]\). The variation heads and results (including the original head) are listed in Table 9.5.
Table 9.5: Closely related variation heads of
{originalPhone:2, objectType:phone, relationThreshold:-2}
Direct Manipulation - The Hysterical Space From Selected Heads

The user may see two (or many) interesting results with the desire to discover new results that share properties from these interesting results. The most likely way to accomplish this is to mix input parameters from both results in order to create new inputting parameters, and then use these new parameters to get results. In our notion, we need to create variation heads based on selected variation heads.

For example, from Table 9.1, we see two interesting results: one is at row 2, column 4, and the other is at row 4, column 3. Variation heads and their indexes associated with these two results are:

\[
[1,0,3] \{\text{originalPhone:2, objectType:phone, relationThreshold:-2}\} \text{ and }\ [3,1,2] \{\text{originalPhone:5, objectType:tower, relationThreshold:3}\}.
\]

Then the variation heads mixed from these two heads are:

\[
\{\text{originalPhone:2, objectType:phone, relationThreshold:3}\} \\
\{\text{originalPhone:5, objectType:phone, relationThreshold:-2}\} \\
\{\text{originalPhone:2, objectType:tower, relationThreshold:3}\} \\
\{\text{originalPhone:5, objectType:tower, relationThreshold:-2}\} \\
\{\text{originalPhone:2, objectType:tower, relationThreshold:-2}\} \\
\{\text{originalPhone:5, objectType:phone, relationThreshold:3}\}
\]

Including the two original result, these new results could be seen in Table 9.6.

To generalize from this, a possible interaction and underlying system procedure to obtain all results related to these chosen results could be:

- The user selects two (or several) results through the GUI with a pointing device.
- From these results, the system identifies the variation heads \( v_0, \ldots, v_n \) and their indices in the variation space \( V \).
- The system creates the hysterical (Section 6.5.5) space \( H \) from these heads \( v_0, \ldots, v_n \).
- Based on \( H \), the system generates and displays results in the GUI.
Table 9.6: Hysterical space created from selected heads
9.2 Use Case B: VAST 2009 Mini Challenge 2 - Criminal Ring

9.2.1 The Problem

The IEEE VAST 2009 conference posted a mini challenge to solve a social network and geo-spatial problem (VAST, 2009). The data provides a bigger network with about 6,000 people and connections among the people. There is a criminal ring which existed in the social network. The structure (connections among people) of the criminal ring is known. This challenge pushes us to identify the structure (among two possibilities) and identify the people (and their roles) in the criminal ring (VAST, 2009).

Employees of an embassy used a social networking tool “Flitter” to communicate with friends and colleagues. The Flitter network may have provided a connection to a criminal ring that may have recruited an employee. The challenge provides the Flitter data for analysis.

Data (people and connections) are provided by two tab-delimited tables: one describing entities (i.e., a Flitter nickname, a city, or a country) and the other containing links. Links are two ways to establish user-to-user connections. The format of these two tables is found in Figure 9.3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Type</th>
<th>ID1</th>
<th>ID2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>@Arthur</td>
<td>person</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>@Joe</td>
<td>person</td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>@Kevin</td>
<td>person</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>@Jerry</td>
<td>person</td>
<td>4</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Kannvic</td>
<td>city</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Flovania</td>
<td>country</td>
<td>11</td>
<td>21</td>
</tr>
</tbody>
</table>

**Figure 9.3:** Example data of the VAST 2009 mini challenge 2.

Currently we know that an employee communicated with his/her handler(s) (a contact from the criminal network) through Flitter. Also, we know that the social network is in one of the following two structures:

“A. The employee has about 40 Flitter contacts. Three of these contacts are his "handlers", people in the criminal organization assigned to obtain his cooperation. Each of the handlers probably has between 30 to 40 Flitter contacts and shares a common middle man in the organization, who we have code-named Boris. Boris maintains contact with
the handlers, but does not allow them to communicate among themselves using Flitter. Boris communicates with one or two others in the organization and no one else. One of these contacts is his likely boss, who we’ve code-named the Fearless Leader. The Fearless Leader probably has a broad Flitter network (well over 100 links), including international contacts."

"B. The employee has about 40 Flitter contacts. Three of these contacts are his "handlers," people in the organization assigned to obtain his cooperation. Each of the handlers likely has between 30 to 40 Flitter contacts, and each probably has his or her own middle man in the organization, who we’ve code-named Boris, Morris, and Horace. It is probable that the middle men will not allow the handlers to communicate amongst themselves using Flitter. Each of the middle men probably communicate with one or two others in the organization, and no one else. One of the contacts for all of the middle men is the head of the organization, Fearless Leader. Fearless Leader has a broad Flitter network (well over 100 links) including international contacts" (VAST, 2009).

With this data, we were asked to identify the structure of the criminal ring and to find out the people and their roles in the network.

9.2.2 The Solution

In this section we will examine the social network structure A. Flitter contacts are CZ-Saw entities. The command commonRelated is essential here to search for entities that are related to a number of “at least” (lower boundary) and “at most” (upper boundary) entities.

The Parametric Model

Based on below facts given by the social network structure A, we have setup constraints among nodes in the graph 9.4.

- Each of the handlers probably has between 30 to 40 Flitter contacts: We make two source nodes handlerLower and handlerUpper with the initial values of 30 and 40. The child node handlerA contains everybody that have handlerLower to handlerUpper contacts.

\[
\text{handlerA} = \text{commonRelated}(\text{all, "person", handlerLower, handlerUpper})
\]

all’s value is for all people in the social network.
• Share a common middle man in the organization, who we have code-named Boris: handlerA has a child node potentialBorisB. potentialBorisB contains people who contact at least three people in node handlerA:
  potentialBorisB = commonRelated(handlerA, "person", 3)

• Boris communicates with one or two others in the organization and no one else: the node potentialBorisA contains people that have at most borisUpper contacts. borisUpper is a small number like 4 or 5 - three handlers plus one or two others.
  potentialBorisA = commonRelated(all, "person", 1, borisUpper)

• Boris node contains people that are both in potentialBorisA and potentialBorisB:
  Boris = intersection(potentialBorisA, potentialBorisB)

• All Boris’s contacts are in the node borisContact, in which should contain the handler and the leader.
  borisContact = relatedNodes("person", Boris)
  The intersection of borisContact and handlerA makes the node handler:
  handler = intersection(borisContact, handlerA)

• Employee . . . Three of these contacts are his handlers: People (potentialEmployee1) contains people that have at least three contacts in handler:
  potentialEmployee1 = commonRelated(handlers, "person", 3)

• The employee has about 40 Flitter contacts: potentialEmployee2 contains people that have about 40 contacts (could be a range e.g., 35 45, which can be assigned with two nodes employeeUpper and employeeLower):
  potentialEmployee2 = commonRelated(all,"person",employeeLower, employeeUpper)
  Intersection of potentialEmployee1 and potentialEmployee2 makes the node employee:
  employee = intersection(potentialEmployee1, potentialEmployee2)

• Fearless Leader probably has a broad Flitter network (well over 100 links): The node potentialLeader contains people that have at least 100 (leadLower) contacts:
  potentialLeader = commonRelated(all, "person", leadLower)

• One of these contacts (of Boris) is his likely boss: Intersection of potentialLeader and borisContact makes the node leader:
  leader = intersection(potentialLeader, borisContact)

• To visually examine connections, we need to identify relations (nodes rEmployeeHandler, rHandlerBoris, rBorisLeader) among nodes leader, Boris, handler and employee:
  rEmployeeHandler = relations(employee, handler)
Figure 9.4: The parametric model for the 2009 social network challenge
{employeeLower:40, employeeUpper:40, handlerLower:30, handlerUpper:40, borisUpper:5, leaderLower:100}

Figure 9.5: Using CZSaw to solve the VAST 2009 social network challenge

The Space of Variation Heads and Results

Based on this model, we have 6 source nodes:
{employeeLower, employeeUpper, handlerLower, handlerUpper, borisUpper, leaderLower}.

They are all single property nodes. From the question, we have initial values for these nodes. Written in a tuple, the values are: {40, 40, 30, 40, 5, 100}. When we apply these values onto CZSaw, we drive CZSaw to display groups of people in its list view, like Figure 9.5. The bottom half of the screenshot is a graph showing connections among people. This graph shows four groups of people, from left to right, where each contains suspect(s) of leader, Boris (middle man), handler, and employee. Edges among these people show the Flitter connections. People with non-connections may be filtered out visually. We can see under the Employee group that there is only one person @schaffter, who can be determined to be the suspect embassy employee. By following its edges, we see three people, @pattersson, @reitenspies, and @kushmir, who should be these three handlers. By
following the edge to the group which contains Boris, we see that Boris’ ID is \(@good\) and the leader’s ID is \(@szemeredi\). This result perfectly matches the answer published by this mini challenge committee (VAST, 2009).

However, in the real world, these numbers may keep on changing; members in Flitter may grow or a member may change his/her contacts. Also, some information (e.g., the numbers of contacts) may not be accurate enough. Therefore, we should give analysts the ability to investigate alternative suspects by using different values. For example, we may want to expand the range of suspects of employees, wondering: What if the employee has 35 or 45 contacts? Or, for the range of Boris, what if he has more than 2 other contacts (allowing him at most to have 6 or maybe 10 contacts)? We can make variation spaces for these source nodes to check for suspects with different connection parameters. In these spaces, the parameter changes from the most restricted number (searching in the narrowest range) to a more generous number (searching in a wider range).

\[
\begin{align*}
\text{employeeLower:} & \{40, 37, 30\} \\
\text{employeeUpper:} & \{40, 43, 50\} \\
\text{handlerLower:} & \{30, 25, 20\} \\
\text{handlerUpper:} & \{40, 45, 50\} \\
\text{borisUpper:} & \{5, 6, 9\} \\
\text{leaderLower:} & \{200, 100, 70\}
\end{align*}
\]

With these individual 1-D spaces, we are able to combine them to test out different social network layouts. Since the numbers of employees’ and handlers’ contacts is a range, we can use an indexing unification to enlarge the searching range of employee and handler. The employee’s search ranges become:

\[
\begin{align*}
\{\text{employeeLower:}40, \text{employeeUpper:}40\}, \\
\{\text{employeeLower:}37, \text{employeeUpper:}43\}, \\
\{\text{employeeLower:}30, \text{employeeUpper:}50\}
\end{align*}
\]

Similarly, the handlers’ search ranges become:

\[
\begin{align*}
\{\text{handlerLower:}30, \text{handlerUpper:}40\}, \\
\{\text{handlerLower:}25, \text{handlerUpper:}45\}, \\
\{\text{handlerLower:}20, \text{handlerUpper:}50\}
\end{align*}
\]
Now the final variation space after the unification of all spaces is
\[
((\text{employeeLower} \cup_{U} \text{employeeUpper}) \cup_{\otimes} (\text{handlerLower} \cup_{U} \text{handlerUpper}) \\
\cup_{\otimes} \text{borisUpper} \cup_{\otimes} \text{leaderLower}).
\]
This variation space contains \(3 \times 3 \times 3 \times 3 = 81\) variations heads. Each of the variation heads contains six properties, with an index length of 4 (e.g., [0,1,2,1]). By applying the variation space into CZSaw, we get results as are shown in Table 9.7.

The scanning of these thumbnails gives us the overview of these results. At first we can look at the number of people listed in each of the four groups. From left to right, these four columns in the thumbnails are a list of people that contains the leader, mid-man Boris, handlers, and the embassy employee. Edges indicate people who communicate through Flitter. By following these edges, we can visually filter out many people in these four groups. Since the suspects are in the list, with more people in the list, there is less chance that we will miss the suspect, but more effort will be required to find the exact suspect.

Since the nodes’ property names (e.g., \text{employeeLower}, \text{handlerUpper}) in this example are pretty long, for convenience and saving space, in the following sections, sometimes I will directly use the value tuple (e.g., \{100, 5, 30, 40, 40, 40\}) for the head. The order is \{leaderLower, borisUpper, handlerLower, handlerUpper, employeeLower, employeeUpper\} to match these four groups of suspects from left to right.
<table>
<thead>
<tr>
<th>LeaderLower: 200</th>
<th>LeaderLower: 100</th>
<th>LeaderLower: 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 40, handlerLower: 30, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 40, handlerLower: 30, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 40, handlerLower: 30, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 45, handlerLower: 25, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 45, handlerLower: 25, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 45, handlerLower: 25, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 50, handlerLower: 20, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7 - Continued
### Table 9.7 – continued from previous page

<table>
<thead>
<tr>
<th>LeaderLower=200</th>
<th>LeaderLower=100</th>
<th>LeaderLower=70</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 50, handlerLower: 20, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 40, employeeLower: 40, handlerUpper: 50, handlerLower: 20, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 40, handlerLower: 30, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 40, handlerLower: 30, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 40, handlerLower: 30, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 45, handlerLower: 25, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 45, handlerLower: 25, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 9.7 - Continued**
### Table 9.7 – continued from previous page

<table>
<thead>
<tr>
<th>LeaderLower=200</th>
<th>LeaderLower=100</th>
<th>LeaderLower=70</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 45, handlerLower: 25, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 50, handlerLower: 20, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 50, handlerLower: 20, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 43, employeeLower: 37, handlerUpper: 50, handlerLower: 20, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 50, employeeLower: 30, handlerUpper: 40, handlerLower: 30, BorisUpper: 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 50, employeeLower: 30, handlerUpper: 40, handlerLower: 30, BorisUpper: 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper: 50, employeeLower: 30, handlerUpper: 40, handlerLower: 30, BorisUpper: 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7 - Continued
Table 9.7 – continued from previous page

<table>
<thead>
<tr>
<th>LeaderLower=200</th>
<th>LeaderLower=100</th>
<th>LeaderLower=70</th>
</tr>
</thead>
<tbody>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:45, handlerLower:25, BorisUpper:5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:45, handlerLower:25, BorisUpper:6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:45, handlerLower:25, BorisUpper:9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:50, handlerLower:20, BorisUpper:5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:50, handlerLower:20, BorisUpper:6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>employeeUpper:50, employeeLower:30, handlerUpper:50, handlerLower:20, BorisUpper:9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7: All possible social networks
Table 9.8: Closely related results by expanding the parameter range
9.2.3 Direct Manipulation - Closely Related Results

Similar to the above section of Challenge 2008, we can study closely related results. The most restricted searching range

\{
leaderLower : 200, borisUpper : 5, handlerLower : 30, handlerUpper : 40, employeeLower : 40, employeeUpper : 40
\}

gives us a clear solution for this challenge (the top left screenshot of Table 9.7). With the parameters changing to expand the range, the number of suspects in each group increases (Table 9.7 and 9.10). We may want to check what happens if we loosen these restrictions a little bit. We can select this result and check for neighboring results (Table 9.8).

9.2.4 Direct Manipulation - Studying the Effect of a Change of Parameters

We can study the effect of a change of only one or several parameters. To do this, we can use the slice function to get a sub-space from the space generated by Cartesian Unification. For example, to check for the effect of borisUpper (at the 3rd element of the index) and the range of employee contacts (employeeLower and employeeUpper) while other parameters remain fixed (e.g., using the first value in its space), we can get a slice of space by using an index \([\bot, 0, \bot, 1]\) which gives us all variation heads that have an index like \([m, 0, n, 1]\) (m, n changes for each variation head). The sliced variation space is listed in Table 9.9.

Table 9.9 reflects the results of this sliced variation space.

To avoid the transition between working with the variation head and working visually within GUI, the above slice call can be implemented through interaction. The procedure could be:
CHAPTER 9. CZSAW USE CASES 215

Table 9.10: Studying the effects of changing three parameters

- Picking up two results from the original results
- The system looks up the variation heads $v, w$ and their indices $I, J$ in the variation space $V$. Indices $I, J$ have a length of $n$, as $[i_0, i_1, \ldots, i_n]$ and $[j_0, j_1, \ldots, j_n]$.
- Make a new index $K$ from $I, J$ by:

```java
for (m=0; m<n; m++) {
    if (i_m != j_m) {
        k_m = \perp;
    } else {
        k_m = i_m;
    }
}
K = [k_0, k_1, \ldots, k_n];
```

- Use index $K$ for slice $V$, get a new space $W$, and then get the new set of results.

The example of Table 9.10 can be done by:
- Pick two results at row 1 / column 2 and row 11 / column 2.
- The system retrieves the associated variation head and index, as
• The system calculates the slicing index $K = [\bot, 0, \bot, 1]$ and gets results.

• Display results as Table 9.10.

### 9.2.5 Direct Manipulation - Order Type Unification

Subsumption and unification of order types (Section 6.2.2) can be used here to enhance the analysis process. According to the meanings of node values, the information carried by the node can be measured. In this closed Flitter social network, people only contact other people within Flitter. The group of people who have at least 40 contacts is a subset of the group with 30 contacts (a person who has at least 40 contacts must have at least 30 contacts, but a person having at least 30 contacts may not have more than 40 contacts).

To allocate a suspect in a group, the smaller the group, the easier to find the suspect. Therefore, we say the sentence “a person has at least 40 contacts” contains more information than the sentence “a person has at least 30 contacts,” because the former sentence provides a smaller and more accurate range for which to search. In this case, the integer number of contacts is the ascending integer - $a \sqsubseteq b$ if $a \leq b$. Vice versa, “a person has at most 5 contacts” contains more information than “a person has at most 10 contacts,” because a person who has 5 or less contacts must have less than 10 contacts. In this case, the integer number of contacts is descending integer - $a \sqsupseteq b$ if $a \geq b$.

In this challenge, nodes with values of ascending integers are: $\text{employeeLower}$, $\text{handlerLower}$, and $\text{leaderLower}$. Nodes with values of descending integers are: $\text{employeeUpper}$, $\text{handlerUpper}$, and $\text{borisUpper}$. As for unification,

$$\{\text{employeeLower}:30\} \sqcup \{\text{employeeLower}:40\} \Rightarrow \{\text{employeeLower}:40\}$$

$$\{\text{borisUpper}:5\} \sqcup \{\text{borisUpper}:6\} \Rightarrow \{\text{borisUpper}:5\}$$

If we randomly choose two (or several) results from Table 9.7, say the corresponding variation heads are \{50, 30, 40, 30, 6, 70\} and \{43, 37, 45, 25, 5, 100\}, we can unify all
Table 9.11: Unifying two chosen results
Table 9.12: A Cartesian square of selected results
chosen results to make one (Table 9.11). In the unified results, the list of suspects are much shorter, which makes it much easier to identify suspects.

We can also use Cartesian Square or Cartesian Power (Section 6.4.6) to generate results from chosen results (if more than two). In Table 9.12, I selected 4 results, then used Cartesian Square to generate a space with 6 results.

9.3 Use Case C: VAST 2010 Mini Challenge 1 - Firearm Dealing

9.3.1 The Problem

The mini challenge 1 of VAST 2010 is intended to investigate firearms dealing activities from 100 text reports. To make this challenge work in CZSaw, we conducted entity extraction (through computer or manually by user) from these text reports. Entity types we extracted are people, location, date, and organization.

Through scanning these documents, we can see that a conference happened in April, 2009 at Dubai, UAE. Many people (and organizations) met in this conference to arrange firearm transactions. Uncovering the details of this conference (who meets who at which date) is important.

9.3.2 The Solution

We created a simple parametric model to search for entities related to a date (Figure 9.6). In this model, the node day contains a set of days (the set may only has one day). relatedEntities contains all related entities of day. entityType defines the type of entities in relatedEntities. Thus, we can find out the details of the conference by finding out who, which location, which organization, or which documents are related to a day (or several days) in April 2009.

From the extracted entities, possible days are April 15, April 16, . . . , April 23. But there is also a date entity called April that may be related to this conference. Since CZSaw allows a node’s value to be a set of entities, we choose to assign day to a set that contains only one day. The space of day is:

\{\{April 15\}, \{April 16\}, \ldots, \{April 22\}, \{April 23\}, \{April\}\}

The space of entityType is:

\{location, people, organization, report\}
Figure 9.6: The parametric model and one state of finding entities related to the Dubai conference
CHAPTER 9. CZSAW USE CASES

The final variation space is $day \sqcup entityType$, and the CZSaw results are located in Table 9.13. Since there are only two nodes in these variation heads, the tabular layout is simple: rows show different values for $days$, and columns show different values for $entityType$.

From these screenshots, we can see that April 18 is related to more entities than are other days. That day could be an important date that is worth further investigation.

9.3.3 Direct Manipulation - A Combined Graph

We have seen graphs for each single day. Our formalism allows us to combine these graphs. Section 6.5.4 gives us the method of unifying variation heads along one dimension, which allows us to see one kind of entity related to all of these April days by unifying variation heads along the $day$ dimension.

The value of the node $day$ is a set of days. We define the unification of two $day$ nodes as the set union of all days in both nodes.

\[
\begin{align*}
1. day_0 &= \{d_{00}, d_{01}, \ldots, d_{0m}\} \\
2. day_1 &= \{d_{10}, d_{11}, \ldots, d_{1n}\} \\
3. day_0 \sqcup day_1 &= \{d_{00}, d_{01}, \ldots, d_{0m}, d_{10}, d_{11}, \ldots, d_{1n}\}
\end{align*}
\]

The user can choose two results from Table 9.13 that differ only in $day$, for example, row 1/column 1 and row 3/column 1. The system will examine the chosen results, look up their index, and find out that these chosen results are different on the index of the first dimension for $day$. CZSaw can then prompt and allow the user to choose to unify results along this $day$ dimension. Thus we will get four new variation heads, with each creates a graph that show different types of related entities to all April days (Figure 9.7, 9.8, 9.9, and 9.10).

From the first screenshot to the last screenshot, we can see location, people, reports, and organizations related to all April days. The location screenshot shows that Dubai connects to most days, which indicates that the conference was happening in Dubai, UAE. The people screenshots show that the busiest days are from April 16 to 18. People are also clustered by different days. By reading through all of the reports in the third screenshots, we will obtain a detailed understanding of the conference. But from all of these days,
<table>
<thead>
<tr>
<th>April 15</th>
<th>location</th>
<th>person</th>
<th>report</th>
<th>organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>April</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.13: Entities related to a single day of April
CHAPTER 9. CZSAW USE CASES

{day:{April 15, April 16, ..., April 23, April}, entityType:Location}

Figure 9.7: Locations related to all April days

{day:{April 15, April 16, ..., April 23, April}, entityType:Person}

Figure 9.8: People related to all April days
Figure 9.9: Reports related to all April days

Figure 9.10: Organizations related to April days
there are not many organizations mentioned (only two: Sana and Lashkar-e-Jhangvi). There were mostly person to person meetings.

9.4 Summary

In this chapter, I have described several ways of employing the formalism in CZSaw in order to handle alternatives by using analytical tasks and data sets from the VAST challenge contests. Applying variation heads in a variation space onto CZSaw’s dependency graph will generate a space of results which contains many varied visualizations. To create a variation space, the user simply assigns a series of values to each parameter (one inputting node in the dependency graph) and uses a simple indexing unification and Cartesian unification to create the space. Thus we are able to provide many results to the system. Then the system developer can design an interface to let the analyst see many different visualizations at once, and then to inspect and compare them.

This formalism can further enrich the user’s interaction over the generated result space. By picking one to several results from the space, the analyst is able to:

- Find closely related results
- Generate a series of results that have mixed parameters from chosen ones (blending)
- Study how changes of one (or several) parameter may affect the results
- Use the order type unification to generate results that combine information from chosen ones.
- Combine several results (simple graphs) into one complex result (a network).

This formalism gives us a start from which to generate and interact with alternatives. Other than the proposed interactions, I believe there are many more operations we can do over the space.
Chapter 10

Conclusion

10.1 The Same Problem Again

Recently I made a mistake similar to that of *Selecting the first answer that appears “good enough”* (George, 1980) while I was working on the VAST 2011 challenge. The task of the VAST 2011 mini challenge 1 is to identify the spread of an epidemic. It requires that we find the origins and transmission methods of a dramatic increase of a flu-like disease taking place in a major metropolitan area. The main dataset has more than a million blog records. Each record contains the blogger’s ID, time, location (longitude and latitude), and a short text message. Many of these blogs record the bloggers’ feeling and symptoms (e.g., fever, chills, nausea, vomiting, and diarrhea). We (a research group in Purdue University) developed the *MobileAnalymator* (Mobile Analysis Animator) to allow users to run the geography-temporal analysis tasks on a handheld computer (e.g., tablets like Motorola Xoom) and within a Web browser. This tool won an award for the “Novel Extension of Visual Analytics to Mobile Devices.” As was stated in reviewers’ comments, “The ability for this to be run on a mobile device allows for it to be used in a real-time environment.”
CHAPTER 10. CONCLUSION

Figure 10.1: MobileAnalyMator: Blogs highlighted by different keywords

a: blogs with flu symptoms

b: blogs with stomach symptoms
The full answer of this challenge is with double sickness outbreaks, where one is transmitted by air and the other is waterborne. We accurately detected the airborne one, but totally neglected the waterborne disease. After the reviewers pointed out our mistake, I went back to check where we went wrong and discovered the problem. While we studied part of the symptoms, we totally ignored the waterborne related symptoms. In the challenge’s task description, the full set of keywords listed are: fever, chills, sweats, aches and pains, fatigue, coughing, breathing difficulty, nausea and vomiting, diarrhea, and enlarged lymph nodes. In our original process of analysis, we only used such keywords as “fever, breath, chill, fatigue, coughing” (Figure 10.1a). These keywords are only related to the airborne sickness. However, using an analysis procedure similar to that for detecting the airborne, but by just replacing these keyword to “nausea, vomiting, diarrhea, lymph,” the MobileAnalyymator is able to detect the waterborne sickness (Figure 10.1b).

My mistake here is simple. I found one “good enough” answer, but I never should have stopped without checking other possible answers. Analysts will experience the same problem again and again. This is frequently due to the inexperience of an analyst, to the limited time and resources, or simply to human beings’ limitation on the cognitive load. Therefore, to help an analyst discover the whole story, the ability to help that analyst handle alternatives is necessary in the application level.

10.2 Review of the Journey

The concept of general alternatives may be too broad (and too complex) to study. This current research is directly motivated by the development and use of a visual analytic tool CZSaw, and it focuses on the variations of input nodes of the parametric model (the representation of propagation-based parametric design system).

A series of users’ interactions can be seen to interactively create a solution for the analysis problem. CZSaw captures these users’ interactions and employs a parametric model (CZSaw dependency graph) to represent the solution. In Chapter 2, I introduced how the current version of CZSaw works, through the process of capturing users’ interactions, building transaction scripts, and constructing the dependency graph. Changing one node’s value will cause the propagation mechanism to update the values of all related nodes in the graph. Thus, it is very easy for the analyst to reuse the solution for different scenarios by assigning different values to inputting nodes. But sometimes there are too
many scenarios to test, leaving us to ask if there is a better way to handle these scenarios. This question directly leads to my research: a formalism to handle variations in a parametric model, which includes definitions of the concepts of variation head and variation space, basic operation of subsumption and unification, and algorithms including indexing unification and Cartesian unification.

In the area of intelligence analysis, many times the analysts make poor decisions due to the limitation of not being able to consider alternatives in a broad enough manner. One of the key tasks that a VA system should support is to help analysts explore alternatives. In Chapter 3, I reviewed several systems in order to see how these may assist the user working with alternatives under their context. These systems include three visual analytic systems (Jigsaw, VisTrails, Tableau, and GeoTime) and the photo edition application, Adobe’s Photoshop. In most of these systems, the user has to manually explore alternatives, which may easily lead the user to fall into the problem of "satisfying" (chose the first answer that satisfied the situation) (Moore, 2007). Some systems (e.g., Tableau and VisTrails) allow the user to change parameters, where the system uses these parameters to automatically construct the visualization. In Photoshop, an experienced user can combine several features (e.g., history tool, snapshot, and layers) to manage variations of a graphic design.

Some systems (or research projects) were designed to handle alternatives. Some projects focus on the user interface; some focus on the underlying representation. In Chapter 4, my study reports on representations of alternatives in three different domains: design space exploration, product family design, and software configuration management.

After establishing the research methodology (Design Science Research), I started to design the formalism in Chapters 6 and 7. Since the concept of alternatives is too broad and general, I chose to focus the research on a narrower topic: variations (similar solutions with different parameters). Chapter 6 first provides the basic object, variation head, and two operations, subsumption and unification. Then I defined a collection of variation heads as a variation space. I define the structure of the space as an indexed array, where each variation head has a corresponding index in the space. Thus, we can easily navigate through the space by following the index. Spaces can be unified to make a more complex space. In chapter 7, I mapped our concept of variation heads and variation spaces onto relational algebra. Based on such mapping, I further brought some ideas to enrich the
formalism.

Chapter 8 and 9 are demonstration and evaluations of the formalism. In Chapter 8, I simulate the variation spaces and operations using node grids in a 3-D geometry space. In this simulation, a node can have up to four properties: X, Y, Z axis values, and the color. Thus, we can visually inspect the variation heads, variation spaces, the algorithms, and computed results.

In Chapter 9, I demonstrated several possible ways of employing the formalism to support interaction and analysis by using three samples of VAST challenges. Other than allowing the user to see many results at once, the formalism can further enrich the user’s interaction over the generated result space. By picking one to several results from the space, the user is able to find closely related results, study the effect of parameter changes, or combine simple graphs into a complex network.

10.3 Achievement and Potentials

Unlike some systems that can use AI algorithms to search for a optimum result (e.g., product family design), this research focuses on systems where the user has to subjectively judge the final results in order to select a satisfactory answer or make the right decision. To maximally avoid bias and omission, the system should provide the user with the ability to examine all possible solutions. Our formalism is designed to help the user construct and interact with multiple inputs, thus enabling the user to explore and compare many solutions.

Within our formalism, the inputs (the variation space) are separated from the outputs (the result space). Through interactions with the variation space, the user can either obtain the full result space by applying the full variation space onto the parametric model or by selectively expanding and exploring the result space through specific interactions with the variation space.

Indices of variation heads in a space can help a user to navigate through the space. The order of indices may encode the user’s preference. The multi-dimensional index allows the user to iterate all of the variation heads and examine all of the possible results freely along dimensions. In addition, by providing an index, the user can create a sub-space which contains similar variation heads or study changes of one or several parameters.
10.3.1 Supporting Interface and Operations of Alternatives

This formalism can satisfy the three design principles defined by Lunzer and Hornbæk (2008).

**Setting up multiple independent scenarios that exist at the same time.** This involves applying one variation space onto a parametric model to create a space of results.

**Viewing those scenarios side by side.** The user can certainly choose several variation heads (e.g., interaction examples from Chapter 9) to get and compare results.

**Making changes to many scenarios in parallel.** Through interacting with the current variation space, the user can generate a new space and thus get new results.

Combined with the parametric model, this formalism can also support the necessary alternative operations as defined by Terry et al. (2004).

**Before a command is invoked:** The user is able to see the variation space (the inputs), and hence realizes that the current state of the problem will necessitate exploring a number of alternatives.

**While interacting with the command:** The propagation mechanism of the parametric model can instantly reflect the change of inputs. Also, the user can interactively construct a new variation space to explore alternatives.

**After a command has been applied:** The index set of the variation space provides an easy way for the user to navigate in the space going either forward or back. The user can simply return to the previous variation head through tracing the index.

Merge is one of the most important operations for the software configuration management system Git (and other version control systems, like SVN). It may also be very useful for other systems in order to obtain a new alternative based on two existing ones. The unification operation proposed in this formalism allows the user to unify variation heads, especially the order type unification, providing an easy way for the user to combine the simple information carried by several inputs (variation heads) in order to produce more complex scenarios (Section 9.3.3).
10.3.2 Supporting Analysis of Competing Hypotheses

Heuer (1999) proposed analysis of competing hypotheses (ACH) as a tool to aid judgment by evaluating multiple competing hypotheses. Complex intelligence issues often require the analyst to choose among several alternative hypotheses. ACH helps the analyst to avoid bias and overcome some of the cognitive limitations on intelligence analysis.

The ACH process comprises eight steps (Heuer, 1999):

1. Identify possible hypotheses. This steps requires the analyst to generate all possible hypotheses while eliminating redundant and irrational ones.

2. List evidences and arguments. Such evidences should include all possible factors that might affect the hypotheses.

3. Create a matrix with hypotheses across the top and evidence down the side. From row to row, identify each evidence to see if it is consistent, inconsistent, or irrelevant to the hypothesis.

4. Refine the matrix by removing irrelevant evidences, collect additional evidences, and reconsidering the hypotheses.

5. Draw tentative conclusions according to the relative likelihood of each hypothesis.

6. Analyze how the conclusion would be affect if critical evidences are wrong, misleading, or subject to different interpretation.

7. Report conclusions. Discuss both the conclusions and other hypotheses that were tested and rejected.

8. Identify milestones for future observation because analytical conclusions are always tentative.

Our proposed formalism can help to generate hypotheses and organize evidences. Within the context of parametric model, for a given problem, the result generated by a model with a set of parameters is a hypothesis. Results generated by different models (alternatives) and different parameter settings of each model (variations) comprise the whole set of competing hypotheses. For example, in the social network challenge of VAST 2009, a hypothesis is a social network graph with each role identified. The social
network could carry different structures, and the person for each role may be different, which may result in a large number of possible criminal networks.

A parametric model is created based on the problem and data. The data contains the evidence. In the VAST 2009 social network challenge, the data include a spreadsheet table of connections, two possible network structures, and possible amount of contacts for each role in each network structure. Some data makes edges of the graph, for example, *Three of these contacts are his (the employee) “handlers”* creates an edge between the employee node and the handler node. Some data fills the value of source nodes; for example, the employee has about 40 Flitter contacts. The number 40 and neighboring numbers are possible values for variation heads. Some data serves as the database for the model to query, like the spreadsheet containing all Flitter connections. In this challenge, each social network structure can be represented by a parametric model, and number of connections defines the parameter of the model. Given a set of parameters, the model produces a social network graph connecting possible suspects. In this problem, these two models makes two families of variations; with each model there is a corresponding variation space. Therefore, constructing the parametric model and organizing the variation spaces are making the list of evidences and arguments. The results generated by the model and insights humans gained from the result are hypotheses.

Since a variation space can be seen as a collection of evidences, the unification is actually combining evidences. When these evidences contain semantically rich order type values, the Cartesian unification will generate a much richer and completed set of evidences. Also, conflicting evidences will be filtered out if the unification result in top. However, this does not mean conflicting evidences will be discarded. Due to the nature of Cartesian unification trying to unify all possible combinations, each original evidence still can make the corresponding hypothesis. These hypotheses with conflicting evidences can then be examined. The unification to top simply means no hypothesis can be supported by both evidences.

Subsumption relationship provides us a way to partially order objects by measuring the carrying information. The subsumption of variation heads also determines the subsumption of corresponding hypotheses, which can help us order the likelihood of hypotheses.

Time consuming is one of several issues that keep ACH been used (Wheaton and
Chido, 2006). When there are numerous hypotheses and a large body of evidence, manually performing the ACH process can be time consuming, which prevent the busy analysts from following the formal process. The propagation mechanism of our parametric model provides an automation way that could an efficient way to speed up the process.

Our current formalism can support variations of one model and can extended to support variation compatible models (different models having the same set of source node). From this prospective, our formalism support to ACH is limited because the problem has to be represented by one parametric model or variation compatible models.

10.3.3 Extending the Formalism

We have distinguished variations and alternatives in Section 6.1.1. This formalism is able to handle variations of a given parametric model. Can we extend this formalism to represent a greater number of general alternatives than merely variations?

The process of creating a parametric model can be seen as a series of parametric models, which are a kind of alternative. In CZSaw, the dependency graph at the current stage is created by executing transactions from the initial until the current transaction. The running of all transactions up until the end will create the final version of the dependency graph, while the running to any transaction in the middle will create a dependency graph in process. Thus, to reach the final model, there is a sequence of dependency graph in the process (the history of the dependency graph).

A new transaction may change the dependency graph in three ways:

- By changing the value of a graphic independent property.
- By adding new nodes. These new nodes may be source nodes or successor nodes.
- By changing the value of a graphic dependent property (the constraint expression). This will change the graph structure of the model.

One way to retrieve a history version of the dependency graph is to reverse these transaction executed after the history version to current version. For example, in the first case, we can change the values of graph independent properties to older values. In the second case, we can remove the added nodes. Our current formalism can perfectly handle the first case by assigning a previous variation head onto the model. The variation space
actually provides the user with more control over these independent properties’ values than simple switching back and forth.

![Diagram](image)

**Figure 10.2:** Parametric models with disabled nodes. Are these models at left equivalent to models at right?

For the second case of adding new nodes, we may disable a node. Suppose that the final model is like that in Figure 10.2a. We can disable a source node or a successor node (Figure 10.2b and c). If one node has been disabled, all of its successor nodes are turned off automatically because there is no way to evaluate the value based on the
constraint expression. In these examples from Figure 10.2b, the disabling of node $I_3$ will automatically disable $B_2, C_1,$ and $D,$ and the disabling of node $B_1$ will automatically disable $C_0, C_1,$ and $D,$ which breaks the model into two unconnected graphs (Figure 10.2d). Figure 10.2b and c will generate the same result, and d and e will generate the same results. Disabling of a node is not equivalent to assigning the term unbound to the node. The term unbound means that there is a value but we do not know that value. In the example of Figure 10.2a , if we replace the value of $C_1$ with unbound, then we do not know where to draw edges to the node $C_1$ because the constraint expression which defines these edges is unknown.

![Diagram](image)

Change the value of node $B_0$. Compared to model a, model b’s $B_0$’s constraint expression does not involve $I_1$.

**Figure 10.3:** Change the value of a graph dependent property

Currently in our formalism, a node can have three states: unbound, top, or assigned with a value. Can we extend the current formalism by adding a new state (disabled) to a node in addition to unbound and top?

In the third case of changing the value (constraint expression) of a graph dependent property, since the constraint expression defines the edges among nodes, such a value change may remove/add edges (Figure 10.3). In this case, these two parametric models are alternatives, rather than variations. Our current formalism only focuses on independent properties. Can we extend the formalism to include the dependent properties, for example, by allowing a successor node to have a space of values (many different constraint expressions)?

In the above examples are some preliminary ideas for extending the formalism to handle a few cases of more general alternatives of the parametric model. At this moment I still cannot provide the essential details (e.g., how to unify two constraint expressions, or how to unify a disabled node with other nodes) for this process.
10.3.4 Generalizations to Other Parametric Systems

For a computing system that provide interfaces for the user to input parameters, I believe this formalism is able to help the user to control the behavior and appearance of the system and its results. Although our formalism was initiated from the parametric models, only the source nodes of the parametric model are involved in the formalism. These source nodes act as parameters for the whole model. For these systems that are controlled by parameters, we can easily map one set of parameters as one variation head, where one parameter is one independent property. In the system for solving VAST 2011 challenge (Figure 10.1), there are several parameters working to control the data in the visualization: filter keywords, highlight keywords, time, and period. The highlight keywords are used to highlight blogs that contain any of the keywords. The given keyword set (fever, chills, sweats, aches and pains, fatigue, coughing, breathing difficulty, nausea and vomiting, diarrhea, enlarged lymph nodes) simply provides the space for highlight keywords. The whole twenty day examination period established the space of time. If I had employed these formalisms on top of the current MobileAnalymator system, I should have been able to detect the full answers.
Appendix A

Appendix: Values of Chapter Eight

Figures

nodeA = a:{x:50}
nodeB = c:{x:50, y:50, z:⊥, c:40}
nodeC = d:{x:50, y:50, z:50, c:100}
vhead = {d:{x:50, y:50, z:50, c:100}, c:{x:50, y:50, z:⊥, c:40}, a:{x:50}}

Figure A.1: A variation head and its 3 nodes in 3D (Figure 8.5)

vspace = {[0]:{a:{x:0, y:0, z:0, c:0}}, [1]:{a:{x:10, y:10, z:10, c:10}}, [2]:{a:{x:20, y:20, z:20, c:20}}, [3]:{a:{x:30, y:30, z:30, c:30}}, [4]:{a:{x:40, y:40, z:40, c:40}}, [5]:{a:{x:50, y:50, z:50, c:50}}, [6]:{a:{x:60, y:60, z:60, c:60}}, [7]:{a:{x:70, y:70, z:70, c:70}}, [8]:{a:{x:80, y:80, z:80, c:80}}, [9]:{a:{x:90, y:90, z:90, c:90}}, [10]:{a:{x:100, y:100, z:100, c:100}}]

Figure A.2: A variation space in 3D (Figure 8.6)

vsaIndices = { [0,0], [0,1], [0,2], [0,3], [1,0], [1,1], [1,2], [1,3], [2,0], [2,1], [2,2], [2,3], [3,0], [3,1], [3,2], [3,3], [4,0], [4,1], [4,2], [4,3]}

Figure A.3: The input of index mapping examples (Figure 8.7)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vsbIndices = { [3,2], [3,3], [3,4], [3,5], [4,2], [4,3], [4,4], [4,5], [5,2], [5,3], [5,4], [5,5], [6,2], [6,3], [6,4], [6,5], [7,2], [7,3], [7,4], [7,5]}

Figure 8.8a

vsclIndices = { [0,0], [0,1], [0,4], [0,5], [1,0], [1,1], [1,4], [1,5], [2,0], [2,1], [2,4], [2,5], [4,0], [4,1], [4,4], [4,5], [5,0], [5,1], [5,4], [5,5]}

Figure 8.8b

Figure A.4: Index shifting examples (Figure 8.8)

vsdIndices = { [0,0], [0,1], [0,2], [0,3], [2,0], [2,1], [2,2], [2,3], [4,0], [4,1], [4,2], [4,3], [6,0], [6,1], [6,2], [6,3], [8,0], [8,1], [8,2], [8,3]}

Figure 8.9c

vseIndices = { [0,0], [0,2], [0,4], [0,6], [2,0], [2,2], [2,4], [2,6], [4,0], [4,2], [4,4], [4,6], [6,0], [6,2], [6,4], [6,6], [8,0], [8,2], [8,4], [8,6]}

Figure 8.9d

Figure A.5: Index distribution examples (Figure 8.9)

vsfIndices = { [0,0,2], [0,1,2], [0,2,2], [0,3,2], [1,0,2], [1,1,2], [1,2,2], [1,3,2], [2,0,2], [2,1,2], [2,2,2], [2,3,2], [3,0,2], [3,1,2], [3,2,2], [3,3,2], [4,0,2], [4,1,2], [4,2,2], [4,3,2]}

Figure 8.10e

vsgIndices = { [2,0,0], [2,0,1], [2,0,2], [2,0,3], [2,1,0], [2,1,1], [2,1,2], [2,1,3], [2,2,0], [2,2,1], [2,2,2], [2,2,3], [2,3,0], [2,3,1], [2,3,2], [2,3,3], [2,4,0], [2,4,1], [2,4,2], [2,4,3]}

Figure 8.10f

Figure A.6: Append and prepend one dimension (Figure 8.10)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vs0Indices = { [1,1], [1,2], [1,3], [1,4], [2,2], [2,3], [2,4], [2,5], [3,3], [3,4], [3,5], [3,6], [4,4], [4,5], [4,6], [4,7], [5,5], [5,6], [5,7], [5,8]}

Figure 8.11a

vs1Indices = { [1,1], [1,2], [1,3], [1,4], [2,5], [3,6], [4,7], [5,8]}

Figure 8.11b

vs1Indices = { [1,4], [2,5], [3,6], [4,7], [5,8]}

Figure 8.11c

vsBRIndices = { [1,1]}

vsTLIndices = { [5,8]}

Figure 8.11d

**Figure A.7:** Boundaries of a 2-D space (Figure 8.11)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vs0Indices = { [0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,0,4], [0,1,0], [0,1,1], [0,1,2], [0,1,3], [0,1,4], [0,2,0], [0,2,1], [0,2,2], [0,2,3], [0,2,4], [0,3,0], [0,3,1], [0,3,2], [0,3,3], [0,3,4], [1,0,0], [1,0,1], [1,0,2], [1,0,3], [1,1,0], [1,1,1], [1,1,2], [1,1,3], [1,1,4], [1,2,0], [1,2,1], [1,2,2], [1,2,3], [1,2,4], [1,3,0], [1,3,1], [1,3,2], [1,3,3], [1,3,4], [2,0,0], [2,0,1], [2,0,2], [2,0,3], [2,0,4], [2,1,0], [2,1,1], [2,1,2], [2,1,3], [2,1,4], [2,2,0], [2,2,1], [2,2,2], [2,2,3], [2,2,4], [2,3,0], [2,3,1], [2,3,2], [2,3,3], [2,3,4]}

Figure 8.12a

vs1Indices = { [0,0,4], [0,1,4], [0,2,4], [0,3,4], [1,0,4], [1,1,4], [1,2,4], [1,3,4], [2,0,4], [2,1,4], [2,2,4], [2,3,4]}

Figure 8.12b

vs1Indices = { [2,3,0], [2,3,1], [2,3,2], [2,3,3], [2,3,4]}

Figure 8.12c

vs1Indices = { [2,3,4]}

Figure 8.12d

Figure A.8: A 3D space for selecting boundary (Figure 8.12)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vxyzIndices = [ [0,0,0], [0,0,1], [0,0,2], [0,0,3], [0,1,0], [0,1,1], [0,1,2], [0,1,3], [0,2,0], [0,2,1], [0,2,2], [0,2,3], [0,3,0], [0,3,1], [0,3,2], [0,3,3], [1,0,0], [1,0,1], [1,0,2], [1,0,3], [1,1,0], [1,1,1], [1,1,2], [1,1,3], [1,2,0], [1,2,1], [1,2,2], [1,2,3], [1,3,0], [1,3,1], [1,3,2], [1,3,3], [2,0,0], [2,0,1], [2,0,2], [2,0,3], [2,1,0], [2,1,1], [2,1,2], [2,1,3], [2,2,0], [2,2,1], [2,2,2], [2,2,3], [2,3,0], [2,3,1], [2,3,2], [2,3,3], [3,0,0], [3,0,1], [3,0,2], [3,0,3], [3,1,0], [3,1,1], [3,1,2], [3,1,3], [3,2,0], [3,2,1], [3,2,2], [3,2,3], [3,3,0], [3,3,1], [3,3,2], [3,3,3]]

Figure 8.13a

txyz1 =={[0,0,2]:{a:{x:0, y:0, z:20}}, [0,1,2]:{a:{x:0, y:10, z:20}}, [0,2,2]:{a:{x:0, y:20, z:20}}, [0,3,2]:{a:{x:0, y:30, z:20}}, [1,0,2]:{a:{x:10, y:0, z:20}}, [1,1,2]:{a:{x:10, y:10, z:20}}, [1,2,2]:{a:{x:10, y:20, z:20}}, [1,3,2]:{a:{x:10, y:30, z:20}}, [2,0,2]:{a:{x:20, y:0, z:20}}, [2,1,2]:{a:{x:20, y:10, z:20}}, [2,2,2]:{a:{x:20, y:20, z:20}}, [2,3,2]:{a:{x:20, y:30, z:20}}, [3,0,2]:{a:{x:30, y:0, z:20}}, [3,1,2]:{a:{x:30, y:10, z:20}}, [3,2,2]:{a:{x:30, y:20, z:20}}, [3,3,2]:{a:{x:30, y:30, z:20}}}]

Figure 8.13b

t2Indices = {[0,2,0], [0,2,1], [0,2,2], [0,2,3], [1,2,0], [1,2,1], [1,2,2], [1,2,3], [2,2,0], [2,2,1], [2,2,2], [2,2,3], [3,2,0], [3,2,1], [3,2,2], [3,2,3]}

Figure 8.13c

t3Indices = {[0,2,1], [1,2,1], [2,2,1], [3,2,1]}

Figure 8.13d

t4Indices = {[1,0,1], [1,1,1], [1,2,1], [1,3,1]}

Figure 8.13e

t5Indices = {[2,2,1]}

Figure 8.13f

**Figure A.9**: Retrieve a subspace by slicing at given index (Figure 8.13)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[
\begin{align*}
\text{vsx} &= \{ [2]: \{ a: \{ x: 20, y: \bot, z: \bot \} \}, [3]: \{ a: \{ x: 30, y: \bot, z: \bot \} \}, [4]: \{ a: \{ x: 40, y: \bot, z: \bot \} \}, [5]: \{ a: \{ x: 50, y: \bot, z: \bot \} \}, [6]: \{ a: \{ x: 60, y: \bot, z: \bot \} \}, [7]: \{ a: \{ x: 70, y: \bot, z: \bot \} \} \}, \\
\text{vsy} &= \{ [4]: \{ a: \{ x: \bot, y: 40, z: \bot \} \}, [5]: \{ a: \{ x: \bot, y: 50, z: \bot \} \}, [6]: \{ a: \{ x: \bot, y: 60, z: \bot \} \}, [7]: \{ a: \{ x: \bot, y: 70, z: \bot \} \}, [8]: \{ a: \{ x: \bot, y: 80, z: \bot \} \}, [9]: \{ a: \{ x: \bot, y: 90, z: \bot \} \} \} \\
\end{align*}
\]

Figure A.10: Inputs of indexing unification (Figure 8.14)

\[
\begin{align*}
\text{vsxy} &= \{ [2]: \{ a: \{ x: 20, y: \bot, z: \bot \} \}, [3]: \{ a: \{ x: 30, y: \bot, z: \bot \} \}, [4]: \{ a: \{ x: 40, y: 40, z: \bot \} \}, [5]: \{ a: \{ x: 50, y: 50, z: \bot \} \}, [6]: \{ a: \{ x: 60, y: 60, z: \bot \} \}, [7]: \{ a: \{ x: 70, y: 70, z: \bot \} \}, [8]: \{ a: \{ x: \bot, y: 80, z: \bot \} \}, [9]: \{ a: \{ x: \bot, y: 90, z: \bot \} \} \} \\
\end{align*}
\]

Figure A.11: Union indexing unification (Figure 8.15)

\[
\begin{align*}
\text{vsxy} &= \{ [4]: \{ a: \{ x: 40, y: 40, z: \bot \} \}, [5]: \{ a: \{ x: 50, y: 50, z: \bot \} \}, [6]: \{ a: \{ x: 60, y: 60, z: \bot \} \}, [7]: \{ a: \{ x: 70, y: 70, z: \bot \} \} \} \\
\end{align*}
\]

Figure A.12: Intersection indexing unification (Figure 8.16)
APPENDIX A: VALUES OF CHAPTER EIGHT FIGURES

\[ \text{vsxy} = \{2[a:x:20, y:z, z:z], 3[a:x:30, y:z, z:z], 4[a:x:40, y:z, z:z], 5[a:x:50, y:z, z:z], 6[a:x:60, y:z, z:z], 7[a:x:70, y:z, z:z] \} \]

**Figure A.13**: Left indexing unification (Figure 8.17)

\[ \text{vsxy} = \{4[a:x:40, y:z, z:z], 5[a:x:50, y:z, z:z], 6[a:x:60, y:z, z:z], 7[a:x:70, y:z, z:z], 8[a:x:80, y:z, z:z], 9[a:x:90, y:z, z:z] \} \]

**Figure A.14**: Right indexing unification (Figure 8.18)

\[ \text{vsxy} = \{2[a:x:20, y:z, z:z], 3[a:x:30, y:z, z:z] \} \]

**Figure A.15**: Left difference unification (Figure 8.19)

\[ \text{vsxy} = \{8[a:x:80, y:z, z:z], 9[a:x:90, y:z, z:z] \} \]

**Figure A.16**: Right difference unification Figure 8.20

\[ \text{vsxy} = \{2,4[a:x:20, y:z, z:z], 2,5[a:x:20, y:50, z:z], 2,6[a:x:20, y:60, z:z], 2,7[a:x:20, y:70, z:z], 2,8[a:x:20, y:80, z:z], 2,9[a:x:20, y:90, z:z], 3,4[a:x:30, y:40, z:z], 3,5[a:x:30, y:50, z:z], 3,6[a:x:30, y:60, z:z], 3,7[a:x:30, y:70, z:z], 3,8[a:x:30, y:80, z:z], 3,9[a:x:30, y:90, z:z], 4,4[a:x:40, y:40, z:z], 4,5[a:x:40, y:50, z:z], 4,6[a:x:40, y:60, z:z], 4,7[a:x:40, y:70, z:z], 4,8[a:x:40, y:80, z:z], 4,9[a:x:40, y:90, z:z], 5,4[a:x:50, y:40, z:z], 5,5[a:x:50, y:50, z:z], 5,6[a:x:50, y:60, z:z], 5,7[a:x:50, y:70, z:z], 5,8[a:x:50, y:80, z:z], 5,9[a:x:50, y:90, z:z], 6,4[a:x:60, y:40, z:z], 6,5[a:x:60, y:50, z:z], 6,6[a:x:60, y:60, z:z], 6,7[a:x:60, y:70, z:z], 6,8[a:x:60, y:80, z:z], 6,9[a:x:60, y:90, z:z], 7,4[a:x:70, y:40, z:z], 7,5[a:x:70, y:50, z:z], 7,6[a:x:70, y:60, z:z], 7,7[a:x:70, y:70, z:z], 7,8[a:x:70, y:80, z:z], 7,9[a:x:70, y:90, z:z] \} \]

**Figure A.17**: Cartesian unification result (Figure 8.21)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ vx = \{(0):\{\text{x:10}\}, (1):\{\text{x:20}\}, (2):\{\text{x:30}\}, (3):\{\text{x:40}\}, (4):\{\text{x:50}\}, (5):\{\text{x:60}\}, (6):\{\text{x:70}\}, (7):\{\text{x:80}\}\} \]

\[ vy = \{(0):\{\text{y:10}\}, (1):\{\text{y:20}\}, (2):\{\text{y:30}\}, (3):\{\text{y:40}\}, (4):\{\text{y:50}\}, (5):\{\text{y:60}\}, (6):\{\text{y:70}\}, (7):\{\text{y:80}\}\} \]

\[ vz = \{(0):\{\text{z:10}\}, (1):\{\text{z:20}\}, (2):\{\text{z:30}\}, (3):\{\text{z:40}\}, (4):\{\text{z:50}\}, (5):\{\text{z:60}\}, (6):\{\text{z:70}\}, (7):\{\text{z:80}\}\} \]

**Figure A.18:** Inputs of indexing unification example (Figure 8.22)

\[ vxy = \{(0):\{\text{x:10, y:10}\}, (1):\{\text{x:20, y:20}\}, (2):\{\text{x:30, y:30}\}, (3):\{\text{x:40, y:40}\}, (4):\{\text{x:50, y:50}\}, (5):\{\text{x:60, y:60}\}, (6):\{\text{x:70, y:70}\}, (7):\{\text{x:80, y:80}\}\} \]

\[ vxz = \{(0):\{\text{x:10, z:10}\}, (1):\{\text{x:20, z:20}\}, (2):\{\text{x:30, z:30}\}, (3):\{\text{x:40, z:40}\}, (4):\{\text{x:50, z:50}\}, (5):\{\text{x:60, z:60}\}, (6):\{\text{x:70, z:70}\}, (7):\{\text{x:80, z:80}\}\} \]

\[ vyz = \{(0):\{\text{y:10, z:10}\}, (1):\{\text{y:20, z:20}\}, (2):\{\text{y:30, z:30}\}, (3):\{\text{y:40, z:40}\}, (4):\{\text{y:50, z:50}\}, (5):\{\text{y:60, z:60}\}, (6):\{\text{y:70, z:70}\}, (7):\{\text{y:80, z:80}\}\} \]

**Figure 8.23a**

\[ vxyz = \{(0):\{\text{x:10, y:10, z:10}\}, (1):\{\text{x:20, y:20, z:20}\}, (2):\{\text{x:30, y:30, z:30}\}, (3):\{\text{x:40, y:40, z:40}\}, (4):\{\text{x:50, y:50, z:50}\}, (5):\{\text{x:60, y:60, z:60}\}, (6):\{\text{x:70, y:70, z:70}\}, (7):\{\text{x:80, y:80, z:80}\}\} \]

**Figure 8.23b**

**Figure A.19:** Indexing unification example (Figure 8.23)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vsIndices = \{ [0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2], [3,0], [3,1], [3,2], [4,0], [4,1], [4,2]\}

vsbIndices = \{ [0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2], [3,0], [3,1], [3,2], [4,0], [4,1], [4,2]\}

Figure 8.24a

vscIndices = \{ [0,0], [0,1], [0,2], [0,3], [0,4], [1,0], [1,1], [1,2], [1,3], [1,4], [2,0], [2,1], [2,2], [2,3], [2,4], [3,0], [3,1], [3,2], [4,0], [4,1], [4,2]\}

Figure 8.24b

vsdIndices = \{ [0,0], [0,1], [0,2], [0,3], [0,4], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2], [2,3], [2,4], [3,0], [3,1], [3,2], [4,0], [4,1], [4,2], [4,3], [4,4], [5,0], [5,1], [5,2]\}

Figure 8.24c

vscIndices = \{ [0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2], [3,0], [3,1], [3,2], [4,0], [4,1], [4,2], [5,0], [5,1], [5,2], [5,3], [5,4], [6,0], [6,1], [6,2], [6,3], [6,4], [7,0], [7,1], [7,2], [7,3], [7,4]\}

Figure 8.24d

Figure A.20: Different ways of combining two spaces (Figure 8.24)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ vsxyz = \{ \text{vsxy} = \{ a : \{ x : 40 \} \} \} \]

\[ \text{vhx} = \{ a : \{ z : 30 \} \} \]

\[ \text{vhyz} = \{ a : \{ x : \perp, y : 40, z : 40 \} \} \]

\[ \text{vsxy} = \{ [2,1]:\{ a : \{ x : 20, y : 10, z : \perp \} \}, [2,2]:\{ a : \{ x : 20, y : 20, z : \perp \} \}, [2,3]:\{ a : \{ x : 20, y : 30, z : \perp \} \}, [2,4]:\{ a : \{ x : 20, y : 40, z : \perp \} \}, [2,5]:\{ a : \{ x : 20, y : 50, z : \perp \} \}, [2,6]:\{ a : \{ x : 20, y : 60, z : \perp \} \}, [2,7]:\{ a : \{ x : 20, y : 70, z : \perp \} \}, \ldots, [7,7]:\{ a : \{ x : 70, y : 70, z : \perp \} \}, [8,1]:\{ a : \{ x : 80, y : 10, z : \perp \} \}, [8,2]:\{ a : \{ x : 80, y : 20, z : \perp \} \}, [8,3]:\{ a : \{ x : 80, y : 30, z : \perp \} \}, [8,4]:\{ a : \{ x : 80, y : 40, z : \perp \} \}, [8,5]:\{ a : \{ x : 80, y : 50, z : \perp \} \}, [8,6]:\{ a : \{ x : 80, y : 60, z : \perp \} \}, [8,7]:\{ a : \{ x : 80, y : 70, z : \perp \} \} \} \]

**Figure 8.25a**

\[ \text{vsxyz} = \{ [2,1]:\{ a : \{ x : 20, y : 10, z : 30 \} \}, [2,2]:\{ a : \{ x : 20, y : 20, z : 30 \} \}, [2,3]:\{ a : \{ x : 20, y : 30, z : 30 \} \}, [2,4]:\{ a : \{ x : 20, y : 40, z : 30 \} \}, [2,5]:\{ a : \{ x : 20, y : 50, z : 30 \} \}, [2,6]:\{ a : \{ x : 20, y : 60, z : 30 \} \}, [2,7]:\{ a : \{ x : 20, y : 70, z : 30 \} \}, \ldots, [7,7]:\{ a : \{ x : 70, y : 70, z : 30 \} \}, [8,1]:\{ a : \{ x : 80, y : 10, z : 30 \} \}, [8,2]:\{ a : \{ x : 80, y : 20, z : 30 \} \}, [8,3]:\{ a : \{ x : 80, y : 30, z : 30 \} \}, [8,4]:\{ a : \{ x : 80, y : 40, z : 30 \} \}, [8,5]:\{ a : \{ x : 80, y : 50, z : 30 \} \}, [8,6]:\{ a : \{ x : 80, y : 60, z : 30 \} \}, [8,7]:\{ a : \{ x : 80, y : 70, z : 30 \} \} \} \]

**Figure 8.25b**

\[ \text{vsxy} = \{ [4,1]:\{ a : \{ x : 40, y : 10, z : \perp \} \}, [4,2]:\{ a : \{ x : 40, y : 20, z : \perp \} \}, [4,3]:\{ a : \{ x : 40, y : 30, z : \perp \} \}, [4,4]:\{ a : \{ x : 40, y : 40, z : \perp \} \}, [4,5]:\{ a : \{ x : 40, y : 50, z : \perp \} \}, [4,6]:\{ a : \{ x : 40, y : 60, z : \perp \} \}, [4,7]:\{ a : \{ x : 40, y : 70, z : \perp \} \} \} \]

**Figure 8.25c**

\[ \text{vsxyz} = \{ [2,4]:\{ a : \{ x : 20, y : 40, z : 40 \} \}, [3,4]:\{ a : \{ x : 30, y : 40, z : 40 \} \}, [4,4]:\{ a : \{ x : 40, y : 40, z : 40 \} \}, [5,4]:\{ a : \{ x : 50, y : 40, z : 40 \} \}, [6,4]:\{ a : \{ x : 60, y : 40, z : 40 \} \}, [7,4]:\{ a : \{ x : 70, y : 40, z : 40 \} \}, [8,4]:\{ a : \{ x : 80, y : 40, z : 40 \} \} \} \]

**Figure 8.25d**

**Figure A.21:** Cartesian unification of a variation head to a space (Figure 8.25)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

248

\[ \begin{align*}
  v_x &= \{(0.0):[a:10], (1.0):[a:20], (2.0):[a:30], (3.0):[a:40], (4.0):[a:50]\} \\
  v_y &= \{(0.0):[a:10], (1.0):[a:20], (2.0):[a:30], (3.0):[a:40], (4.0):[a:50]\} \\
  v_z &= \{(0.0):[a:10], (1.0):[a:20], (2.0):[a:30], (3.0):[a:40]\}
\end{align*} \]

Figure 8.26a

\[ \begin{align*}
  v_{xy} &= \{(0.0):[a:10, y:10], (0.1):[a:10, y:20], (0.2):[a:10, y:30], (0.3):[a:10, y:40], (0.4):[a:10, y:50], (0.5):[a:10, y:60], (1.0):[a:20, y:10], \ldots, (3.5):[a:40, y:60], (4.0):[a:50, y:10], (4.1):[a:50, y:20], (4.2):[a:50, y:30], (4.3):[a:50, y:40], (4.4):[a:50, y:50], (4.5):[a:50, y:60]\} \\
  v_{xz} &= \{(0.0):[a:10, z:10], (0.1):[a:10, z:20], (0.2):[a:10, z:30], (0.3):[a:10, z:40], (1.0):[a:20, z:10], \ldots, (3.3):[a:40, z:40], (4.0):[a:50, z:10], (4.1):[a:50, z:20], (4.2):[a:50, z:30], (4.3):[a:50, z:40]\} \\
  v_{yz} &= \{(0.0):[a:10, y:10], (0.1):[a:10, y:20], (0.2):[a:10, y:30], (0.3):[a:10, y:40], (1.0):[a:20, z:10], \ldots, (4.3):[a:50, y:40], (5.0):[a:60, z:10], (5.1):[a:60, z:20], (5.2):[a:60, z:30], (5.3):[a:60, z:40]\}
\end{align*} \]

Figure 8.26b

\[ \begin{align*}
  v_{xyz} &= \{(0.0,0.0):[a:10, y:10, z:10], (0.0,0.1):[a:10, y:10, z:20], (0.0,0.2):[a:10, y:10, z:30], (0.0,0.3):[a:10, y:10, z:40], (0.1,0.0):[a:10, y:20, z:10], \ldots, (0.5,0.0):[a:10, y:60, z:10], (0.5,1.0):[a:10, y:60, z:20], (0.5,2.0):[a:10, y:60, z:30], (0.5,3.0):[a:10, y:60, z:40], (1.0,0.0):[a:20, y:10, z:10], (1.0,0.1):[a:20, y:10, z:20], (1.0,0.2):[a:20, y:10, z:30], (1.0,0.3):[a:20, y:10, z:40], (1.1,0.0):[a:20, y:20, z:10], \ldots, (3.5,3.0):[a:40, y:60, z:40], (4.0,0.0):[a:50, y:10, z:10], (4.0,0.1):[a:50, y:10, z:20], (4.0,0.2):[a:50, y:10, z:30], (4.0,0.3):[a:50, y:10, z:40], (4.1,0.0):[a:50, y:20, z:10], (4.1,0.1):[a:50, y:20, z:20], (4.1,0.2):[a:50, y:20, z:30], \ldots, (4.5,0.0):[a:50, y:60, z:10], (4.5,1.0):[a:50, y:60, z:20], (4.5,2.0):[a:50, y:60, z:30], (4.5,3.0):[a:50, y:60, z:40]\}
\end{align*} \]

Figure 8.26c

Figure A.22: Cartesian unification of 1-D spaces (Figure 8.26)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[
vxy = \{[0]:\{a:x:10, y:10\}, [1]:\{a:x:20, y:20\}, [2]:\{a:x:30, y:30\}, [3]:\{a:x:40, y:40\}, [4]:\{a:x:50, y:50\}, [5]:\{a:x:60, y:60\}, [6]:\{a:x:70, y:70\}, [7]:\{a:x:80, y:80\}\}
\]

\[
vxz = \{[0]:\{a:x:10, z:10\}, [1]:\{a:x:20, z:20\}, [2]:\{a:x:30, z:30\}, [3]:\{a:x:40, z:40\}, [4]:\{a:x:50, z:50\}, [5]:\{a:x:60, z:60\}, [6]:\{a:x:70, z:70\}, [7]:\{a:x:80, z:80\}\}
\]

\[
\]

**Figure A.23:** Cartesian unification example I - inputs (Figure 8.27)
Figure A.24: Cartesian unification example I - results (Figure 8.28)

vxy = \[(0,0):[a:x:10, y:10], [1,0]:[a:x:20, y:20], [2,1]:[a:x:30, y:30], [3,1]:[a:x:40, y:40], [4,1]:[a:x:50, y:50], [5,1]:[a:x:60, y:60], [6,1]:[a:x:70, y:70]]\)

vxz = \[(0,0):[a:x:10, z:10], [1,0]:[a:x:20, z:20], [2,1]:[a:x:30, z:30], [3,1]:[a:x:40, z:40], [4,1]:[a:x:50, z:50], [5,1]:[a:x:60, z:60], [6,1]:[a:x:70, z:70]]\)

vyz = \[(0,0):[a:y:10, z:10], [1,0]:[a:y:20, z:20], [2,1]:[a:y:30, z:30], [3,1]:[a:y:40, z:40], [4,1]:[a:y:50, z:50], [5,1]:[a:y:60, z:60], [6,1]:[a:y:70, z:70]]\)

Figure 8.29a

vxyz = \[(0,0):[a:x:10, y:10, z:10], [0,1]:[a:x:10, y:20, z:20], [0,2]:[a:x:10, y:30, z:30], [0,3]:[a:x:10, y:40, z:40], [0,4]:[a:x:10, y:50, z:50], [0,5]:[a:x:10, y:60, z:60], [0,6]:[a:x:10, y:70, z:70]]\)

Figure 8.29b
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ vxy = \{ [0]:\{ a: \{ x: 10, y: 10 \} \}, [1]:\{ a: \{ x: 20, y: 20 \} \}, [2]:\{ a: \{ x: 30, y: 30 \} \}, [3]:\{ a: \{ x: 40, y: 40 \} \}, [4]:\{ a: \{ x: 50, y: 50 \} \}, [5]:\{ a: \{ x: 60, y: 60 \} \}, [6]:\{ a: \{ x: 70, y: 70 \} \}, [7]:\{ a: \{ x: 80, y: 80 \} \} \} \]

\[ vxyz = \{ [0,0]:\{ a: \{ x: 10, y: 10, z: 10 \} \}, [0,1]:\{ a: \{ x: 20, y: 10, z: 20 \} \}, [0,2]:\{ a: \{ x: 30, y: 10, z: 30 \} \}, [0,3]:\{ a: \{ x: 40, y: 10, z: 40 \} \}, [0,4]:\{ a: \{ x: 50, y: 10, z: 50 \} \}, [0,5]:\{ a: \{ x: 60, y: 10, z: 60 \} \}, [0,6]:\{ a: \{ x: 70, y: 10, z: 70 \} \}, [0,7]:\{ a: \{ x: 80, y: 10, z: 80 \} \}, [1,0]:\{ a: \{ x: 10, y: 20, z: 10 \} \}, \ldots, [6,7]:\{ a: \{ x: 80, y: 70, z: 80 \} \}, [7,0]:\{ a: \{ x: 10, y: 80, z: 10 \} \}, [7,1]:\{ a: \{ x: 20, y: 80, z: 20 \} \}, [7,2]:\{ a: \{ x: 30, y: 80, z: 30 \} \}, [7,3]:\{ a: \{ x: 40, y: 80, z: 40 \} \}, [7,4]:\{ a: \{ x: 50, y: 80, z: 50 \} \}, [7,5]:\{ a: \{ x: 60, y: 80, z: 60 \} \}, [7,6]:\{ a: \{ x: 70, y: 80, z: 70 \} \}, [7,7]:\{ a: \{ x: 80, y: 80, z: 80 \} \} \} \]

Figure 8.30a

\[ vxyz2 = \{ [0,0,0]:\{ a: \{ x: 10, y: 10, z: 10 \} \}, [1,1,1]:\{ a: \{ x: 20, y: 20, z: 20 \} \}, [2,2,2]:\{ a: \{ x: 30, y: 30, z: 30 \} \}, [3,3,3]:\{ a: \{ x: 40, y: 40, z: 40 \} \}, [4,4,4]:\{ a: \{ x: 50, y: 50, z: 50 \} \}, [5,5,5]:\{ a: \{ x: 60, y: 60, z: 60 \} \}, [6,6,6]:\{ a: \{ x: 70, y: 70, z: 70 \} \}, [7,7,7]:\{ a: \{ x: 80, y: 80, z: 80 \} \} \} \]

Figure 8.30b

**Figure A.26**: Cartesian unification example III (Figure 8.30)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vxyz2 = [{0,0}:a:{x:10, y:10, z:10}], [0,1]:a:{x:20, y:10, z:20}], [0,2]:a:{x:30, y:10, z:30}], [0,3]:a:{x:40, y:10, z:40}], [0,4]:a:{x:50, y:10, z:50}], [0,5]:a:{x:60, y:10, z:60}], [0,6]:a:{x:70, y:10, z:70}], [0,7]:a:{x:80, y:10, z:80}], [1,0]:a:{x:10, y:20, z:10}], ..., [6,7]:a:{x:80, y:70, z:80}], [7,0]:a:{x:10, y:80, z:10}], [7,1]:a:{x:20, y:80, z:20}], [7,2]:a:{x:30, y:80, z:30}], [7,3]:a:{x:40, y:80, z:40}], [7,4]:a:{x:50, y:80, z:50}], [7,5]:a:{x:60, y:80, z:60}], [7,6]:a:{x:70, y:80, z:70}], [7,7]:a:{x:80, y:80, z:80}]

vxyz3 = [{0,0}:a:{x:10, y:10, z:10}], [0,1]:a:{x:20, y:20, z:10}], [0,2]:a:{x:30, y:30, z:10}], [0,3]:a:{x:40, y:40, z:10}], [0,4]:a:{x:50, y:50, z:10}], [0,5]:a:{x:60, y:60, z:10}], [0,6]:a:{x:70, y:70, z:10}], [0,7]:a:{x:80, y:80, z:10}], [1,0]:a:{x:10, y:10, z:20}], ..., [6,7]:a:{x:80, y:80, z:70}], [7,0]:a:{x:10, y:10, z:80}], [7,1]:a:{x:20, y:20, z:80}], [7,2]:a:{x:30, y:30, z:80}], [7,3]:a:{x:40, y:40, z:80}], [7,4]:a:{x:50, y:50, z:80}], [7,5]:a:{x:60, y:60, z:80}], [7,6]:a:{x:70, y:70, z:80}], [7,7]:a:{x:80, y:80, z:80}]

Figure 8.31a

vxyz5 = [{0,0,0,0}:a:{x:10, y:10, z:10}], [1,1,1,1]:a:{x:20, y:20, z:20}], [2,2,2,2]:a:{x:30, y:30, z:30}], [3,3,3,3]:a:{x:40, y:40, z:40}], [4,4,4,4]:a:{x:50, y:50, z:50}], [5,5,5,5]:a:{x:60, y:60, z:60}], [6,6,6,6]:a:{x:70, y:70, z:70}], [7,7,7,7]:a:{x:80, y:80, z:80}]

Figure 8.31b

Figure A.27: Cartesian unification example IV (Figure 8.31)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES


vs2d = [[1,1]:[a:x:10, y:⊥, z:10]], [1,2]:[a:x:10, y:⊥, z:20]], [1,3]:[a:x:10, y:⊥, z:30]], [2,1]:[a:x:20, y:⊥, z:10]], [2,2]:[a:x:20, y:⊥, z:20]], [2,3]:[a:x:20, y:⊥, z:30]], [3,1]:[a:x:30, y:⊥, z:10]], [3,2]:[a:x:30, y:⊥, z:20]], [3,3]:[a:x:30, y:⊥, z:30]], [4,1]:[a:x:40, y:⊥, z:10]], [4,2]:[a:x:40, y:⊥, z:20]], [4,3]:[a:x:40, y:⊥, z:30]], [5,1]:[a:x:50, y:⊥, z:10]], [5,2]:[a:x:50, y:⊥, z:20]], [5,3]:[a:x:50, y:⊥, z:30]], [6,1]:[a:x:60, y:⊥, z:10]], [6,2]:[a:x:60, y:⊥, z:20]], [6,3]:[a:x:60, y:⊥, z:30]]

Figure 8.32a

vs3d = [[1,1,1]:[a:x:10, y:10, z:10, c:10]], [1,1,2]:[a:x:10, y:10, z:20, c:10]], [1,1,3]:[a:x:10, y:10, z:30, c:10]], [2,2,1]:[a:x:20, y:20, z:10, c:20]], [2,2,2]:[a:x:20, y:20, z:20, c:20]], [2,2,3]:[a:x:20, y:20, z:30, c:20]], [3,3,1]:[a:x:30, y:30, z:10, c:30]], [3,3,2]:[a:x:30, y:30, z:20, c:30]], [3,3,3]:[a:x:30, y:30, z:30, c:30]], [4,4,1]:[a:x:40, y:40, z:10, c:40]], [4,4,2]:[a:x:40, y:40, z:20, c:40]], [4,4,3]:[a:x:40, y:40, z:30, c:40]], [5,5,1]:[a:x:50, y:50, z:10, c:50]], [5,5,2]:[a:x:50, y:50, z:20, c:50]], [5,5,3]:[a:x:50, y:50, z:30, c:50]], [6,6,1]:[a:x:60, y:60, z:10, c:60]], [6,6,2]:[a:x:60, y:60, z:20, c:60]], [6,6,3]:[a:x:60, y:60, z:30, c:60]]

Figure 8.32b

Figure A.28: Cartesian unification example V (Figure 8.32)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vsxy = [(1,1):[a:x:10, y:10, z:⊥]], [1,2]:[a:x:10, y:20, z:⊥]], [1,3]:[a:x:10, y:30, z:⊥]], [1,4]:[a:x:10, y:40, z:⊥]], [2,1]:[a:x:20, y:10, z:⊥]], ..., [4,4]:[a:x:40, y:40, z:⊥]], [5,1]:[a:x:50, y:10, z:⊥]], [5,2]:[a:x:50, y:20, z:⊥]], [5,3]:[a:x:50, y:30, z:⊥]], [5,4]:[a:x:50, y:40, z:⊥]], [6,1]:[a:x:60, y:10, z:⊥]], [6,2]:[a:x:60, y:20, z:⊥]], [6,3]:[a:x:60, y:30, z:⊥]], [6,4]:[a:x:60, y:40, z:⊥]]

vsxz = [(1,1):[a:x:10, y:⊥, z:10]], [1,2]:[a:x:10, y:⊥, z:20]], [1,3]:[a:x:10, y:⊥, z:30]], [1,4]:[a:x:10, y:⊥, z:40]], [2,1]:[a:x:20, y:⊥, z:10]], ..., [4,4]:[a:x:40, y:⊥, z:20]], [5,1]:[a:x:50, y:⊥, z:10]], [5,2]:[a:x:50, y:⊥, z:20]], [5,3]:[a:x:50, y:⊥, z:30]], [5,4]:[a:x:50, y:⊥, z:40]]

Figure 8.33a

vs4d = [(1,1,1,1):[a:x:10, y:10, z:10]], [1,1,1,2]:[a:x:10, y:10, z:20]], [1,1,1,3]:[a:x:10, y:10, z:30]], [1,1,1,4]:[a:x:10, y:10, z:40]], [1,2,1,1]:[a:x:10, y:20, z:10]], ..., [5,2,5,4]:[a:x:50, y:20, z:40]], [5,3,5,1]:[a:x:50, y:30, z:10]], [5,3,5,2]:[a:x:50, y:30, z:20]], [5,3,5,3]:[a:x:50, y:30, z:30]], [5,3,5,4]:[a:x:50, y:30, z:40]], [5,4,5,1]:[a:x:50, y:40, z:10]], [5,4,5,2]:[a:x:50, y:40, z:20]], [5,4,5,3]:[a:x:50, y:40, z:30]], [5,4,5,4]:[a:x:50, y:40, z:40]]

Figure 8.33b

vs2du = [(1,1):[a:x:10, y:10, z:10]], [1,2]:[a:x:10, y:20, z:20]], [1,3]:[a:x:10, y:30, z:30]], [1,4]:[a:x:10, y:40, z:40]], [2,1]:[a:x:20, y:10, z:⊥]], ..., [5,4]:[a:x:50, y:40, z:40]], [6,1]:[a:x:60, y:10, z:⊥]], [6,2]:[a:x:60, y:20, z:⊥]], [6,3]:[a:x:60, y:30, z:⊥]], [6,4]:[a:x:60, y:40, z:⊥]]

Figure 8.33c

vs2di = [(1,1):[a:x:10, y:10, z:10]], [1,2]:[a:x:10, y:20, z:20]], [1,3]:[a:x:10, y:30, z:30]], [1,4]:[a:x:10, y:40, z:40]], [2,1]:[a:x:20, y:10, z:⊥]], ..., [5,4]:[a:x:50, y:40, z:40]], [5,1]:[a:x:50, y:10, z:10]], [5,2]:[a:x:50, y:20, z:20]], [5,3]:[a:x:50, y:30, z:30]], [5,4]:[a:x:50, y:40, z:40]]

Figure 8.33d

Figure A.29: Cartesian unifications vs. indexing unifications (Figure 8.33)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ \begin{align*}
\text{vxy} &= \{[0]:\{a: \{x: 0, y: 0\}\}, [1]:\{a: \{x: 10, y: 10\}\}, [2]:\{a: \{x: 20, y: 20\}\}, [3]:\{a: \{x: 30, y: 30\}\}, \\
&\quad [4]:\{a: \{x: 40, y: 40\}\}, [5]:\{a: \{x: 50, y: 50\}\}\}
\end{align*} \]

\[ \begin{align*}
\text{vxz} &= \{[0,0]:\{a: \{x: 0, z: 0\}\}, [0,1]:\{a: \{x: 0, z: 10\}\}, [0,2]:\{a: \{x: 0, z: 20\}\}, [0,3]:\{a: \{x: 0, z: 30\}\}, \\
&\quad [0,4]:\{a: \{x: 0, z: 40\}\}, [0,5]:\{a: \{x: 0, z: 50\}\}, [1,0]:\{a: \{x: 10, z: 0\}\}, [1,1]:\{a: \{x: 10, z: 10\}\}, \ldots, \\
&\quad [4,5]:\{a: \{x: 40, z: 50\}\}, [5,0]:\{a: \{x: 50, z: 0\}\}, [5,1]:\{a: \{x: 50, z: 10\}\}, [5,2]:\{a: \{x: 50, z: 20\}\}, \\
&\quad [5,3]:\{a: \{x: 50, z: 30\}\}, [5,4]:\{a: \{x: 50, z: 40\}\}, [5,5]:\{a: \{x: 50, z: 50\}\}\} \]
\end{align*} \]

Figure 8.35a

\[ \begin{align*}
\text{vxyz} &= \{[0,0]:\{a: \{x: 0, y: 0, z: 0\}\}, [0,1]:\{a: \{x: 0, y: 0, z: 10\}\}, [0,2]:\{a: \{x: 0, y: 0, z: 20\}\}, [0,3]:\{a: \{x: 0, y: 0, z: 30\}\}, \\
&\quad [0,4]:\{a: \{x: 0, y: 0, z: 40\}\}, [0,5]:\{a: \{x: 0, y: 0, z: 50\}\}, [1,0]:\{a: \{x: 10, y: 10, z: 0\}\}, \\
&\quad [1,1]:\{a: \{x: 10, y: 10, z: 10\}\}, \ldots, [4,4]:\{a: \{x: 40, y: 40, z: 40\}\}, [4,5]:\{a: \{x: 40, y: 40, z: 50\}\}, \\
&\quad [5,0]:\{a: \{x: 50, y: 50, z: 0\}\}, [5,1]:\{a: \{x: 50, y: 50, z: 10\}\}, [5,2]:\{a: \{x: 50, y: 50, z: 20\}\}, [5,3]:\{a: \{x: 50, y: 50, z: 30\}\}, \\
&\quad [5,4]:\{a: \{x: 50, y: 50, z: 40\}\}, [5,5]:\{a: \{x: 50, y: 50, z: 50\}\}\} \]
\end{align*} \]

Figure 8.35b

\[ \begin{align*}
\text{vxyz} &= \{[0,0]:\{a: \{x: 0, y: 0, z: 0\}\}, [1,1]:\{a: \{x: 10, y: 10, z: 10\}\}, [2,2]:\{a: \{x: 20, y: 20, z: 20\}\}, \\
&\quad [3,3]:\{a: \{x: 30, y: 30, z: 30\}\}, [4,4]:\{a: \{x: 40, y: 40, z: 40\}\}, [5,5]:\{a: \{x: 50, y: 50, z: 50\}\}\} \]
\end{align*} \]

Figure 8.35c

\textbf{Figure A.30:} Mix a 1-D space with a 2-D space (Figure 8.35)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vxy = \{[0,0]:[a:x0, y0]), [0,1]:[a:x0, y1]), [0,2]:[a:x0, y2]), [0,3]:[a:x0, y3]), [0,4]:[a:x0, y4]), [1,0]:[a:x1, y0]), \ldots, [4,4]:[a:x4, y4]), [5,0]:[a:x5, y0]), [5,1]:[a:x5, y1]), [5,2]:[a:x5, y2]), [5,3]:[a:x5, y3]), [5,4]:[a:x5, y4])\}

vxz = \{[0,0]:[a:x0, z0]), [0,1]:[a:x0, z1]), [0,2]:[a:x0, z2]), [0,3]:[a:x0, z3]), [1,0]:[a:x1, z0]), \ldots, [3,3]:[a:x3, z3]), [4,0]:[a:x4, z0]), [4,1]:[a:x4, z1]), [4,2]:[a:x4, z2]), [4,3]:[a:x4, z3]), [5,0]:[a:x5, z0]), [5,1]:[a:x5, z1]), [5,2]:[a:x5, z2]), [5,3]:[a:x5, z3])\}

Figure 8.36a

vxyz = \{[0,0,0]:[a:x0, y0, z0]), [0,0,1]:[a:x0, y0, z1]), [0,0,2]:[a:x0, y0, z2]), [0,0,3]:[a:x0, y0, z3]), [0,1,0]:[a:x1, y0, z0]), \ldots, [5,2,3]:[a:x5, y2, z3]), [5,3,0]:[a:x5, y3, z0]), [5,3,1]:[a:x5, y3, z1]), [5,3,2]:[a:x5, y3, z2]), [5,3,3]:[a:x5, y3, z3]), [5,4,0]:[a:x5, y4, z0]), [5,4,1]:[a:x5, y4, z1]), [5,4,2]:[a:x5, y4, z2]), [5,4,3]:[a:x5, y4, z3])\}

Figure 8.36b

vxyz = \{[0,0,0]:[a:x0, y0, z0]), [0,1,0]:[a:x1, y0, z0]), [0,2,0]:[a:x2, y0, z0]), [0,3,0]:[a:x3, y0, z0]), [0,4,0]:[a:x4, y0, z0]), [1,0,1]:[a:x1, y1, z0]), \ldots, [2,3,2]:[a:x2, y3, z2]), [2,4,2]:[a:x2, y4, z2]), [3,0,3]:[a:x3, y0, z3]), [3,1,3]:[a:x3, y1, z3]), [3,2,3]:[a:x3, y2, z3]), [3,3,3]:[a:x3, y3, z3]), [3,4,3]:[a:x3, y4, z3])\}

Figure 8.36c

vxyz = \{[0,0]:[a:x0, y0, z0]), [0,1]:[a:x0, y1, z1]), [0,2]:[a:x0, y2, z2]), [0,3]:[a:x0, y3, z3]), [1,0]:[a:x1, y0, z0]), [1,1]:[a:x1, y1, z1]), [1,2]:[a:x1, y2, z2]), [1,3]:[a:x1, y3, z3]), [2,0]:[a:x2, y0, z0]), [2,1]:[a:x2, y1, z1]), [2,2]:[a:x2, y2, z2]), [2,3]:[a:x2, y3, z3]), [3,0]:[a:x3, y0, z0]), [3,1]:[a:x3, y1, z1]), [3,2]:[a:x3, y2, z2]), [3,3]:[a:x3, y3, z3]), [4,0]:[a:x4, y0, z0]), [4,1]:[a:x4, y1, z1]), [4,2]:[a:x4, y2, z2]), [4,3]:[a:x4, y3, z3]), [5,0]:[a:x5, y0, z0]), [5,1]:[a:x5, y1, z1]), [5,2]:[a:x5, y2, z2]), [5,3]:[a:x5, y3, z3])\}

Figure 8.36d

Figure A.31: Mix two 2-D spaces (Figure 8.36)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ \text{vsxy}_0 = \{[0]:\{a: \text{asc 0}, y: \text{desc 15}, c: \text{asc 0}\}, [1]:\{a: \text{asc 10}, y: \text{desc 20}, c: \text{asc 10}\}, [2]:\{a: \text{asc 20}, y: \text{desc 25}, c: \text{asc 20}\}, [3]:\{a: \text{asc 30}, y: \text{desc 30}, c: \text{asc 30}\}, [4]:\{a: \text{asc 40}, y: \text{desc 35}, c: \text{asc 40}\}, [5]:\{a: \text{asc 50}, y: \text{desc 40}, c: \text{asc 50}\}, [6]:\{a: \text{asc 60}, y: \text{desc 45}, c: \text{asc 60}\}\} \]

\[ \text{vsxy}_1 = \{[2]:\{a: \text{asc 25}, y: \text{desc 15}, c: \text{asc 20}\}, [3]:\{a: \text{asc 30}, y: \text{desc 25}, c: \text{asc 30}\}, [4]:\{a: \text{asc 35}, y: \text{desc 35}, c: \text{asc 40}\}, [5]:\{a: \text{asc 40}, y: \text{desc 45}, c: \text{asc 50}\}, [6]:\{a: \text{asc 45}, y: \text{desc 55}, c: \text{asc 60}\}, [7]:\{a: \text{asc 50}, y: \text{desc 65}, c: \text{asc 70}\}, [8]:\{a: \text{asc 55}, y: \text{desc 75}, c: \text{asc 80}\}\} \]

**Figure 8.37a**

\[ \text{vsxy}_2 = \{[2]:\{a: \text{x: desc 20}, y: \text{asc 25}, c: \text{asc 20}\}, [3]:\{a: \text{x: desc 30}, y: \text{asc 30}, c: \text{asc 30}\}, [4]:\{a: \text{x: desc 35}, y: \text{asc 35}, c: \text{asc 40}\}, [5]:\{a: \text{x: desc 40}, y: \text{asc 45}, c: \text{asc 50}\}, [6]:\{a: \text{x: desc 45}, y: \text{asc 55}, c: \text{asc 60}\}\} \]

**Figure 8.37b**

\[ \text{vsxy}_2 = \{[2]:\{a: \text{asc 20}, y: \text{desc 15}, c: \text{asc 20}\}, [3]:\{a: \text{asc 30}, y: \text{desc 25}, c: \text{asc 30}\}, [4]:\{a: \text{asc 35}, y: \text{desc 35}, c: \text{asc 40}\}, [5]:\{a: \text{asc 40}, y: \text{desc 45}, c: \text{asc 50}\}, [6]:\{a: \text{asc 45}, y: \text{desc 55}, c: \text{asc 60}\}\} \]

**Figure 8.37c**

\[ \text{vsxy}_2 = \{[2]:\{a: \text{asc 25}, y: \text{desc 15}, c: \text{asc 20}\}, [3]:\{a: \text{asc 30}, y: \text{desc 25}, c: \text{asc 30}\}, [4]:\{a: \text{asc 40}, y: \text{desc 35}, c: \text{asc 40}\}, [5]:\{a: \text{asc 50}, y: \text{desc 40}, c: \text{asc 50}\}, [6]:\{a: \text{asc 60}, y: \text{desc 45}, c: \text{asc 60}\}\} \]

**Figure 8.37d**

\[ \text{vsxy}_2 = \{[2]:\{a: \text{desc 20}, y: \text{asc 25}, c: \text{asc 20}\}, [3]:\{a: \text{desc 30}, y: \text{asc 30}, c: \text{asc 30}\}, [4]:\{a: \text{desc 35}, y: \text{asc 35}, c: \text{asc 40}\}, [5]:\{a: \text{desc 40}, y: \text{asc 45}, c: \text{asc 50}\}, [6]:\{a: \text{desc 45}, y: \text{asc 55}, c: \text{asc 60}\}\} \]

**Figure 8.37e**

**Figure A.32**: Indexing unification of heads with ascending or descending integers (Figure 8.37)
\textbf{APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES}

\begin{verbatim}
vhx = {a:{x:asc 35}}

vhyz = {a:{x:\perp, y:desc 40, z:40}}

vsxy = {[1,0]:[a:{x:asc 20, y:desc 10}], [1,1]:[a:{x:asc 20, y:desc 30}], [1,2]:[a:{x:asc 20, y:desc 50}], [1,3]:[a:{x:asc 20, y:desc 70}], [2,0]:[a:{x:asc 40, y:desc 10}], [2,1]:[a:{x:asc 40, y:desc 30}], [2,2]:[a:{x:asc 40, y:desc 50}], [2,3]:[a:{x:asc 40, y:desc 70}], [3,0]:[a:{x:asc 60, y:desc 10}], [3,1]:[a:{x:asc 60, y:desc 30}], [3,2]:[a:{x:asc 60, y:desc 50}], [3,3]:[a:{x:asc 60, y:desc 70}], [4,0]:[a:{x:asc 80, y:desc 10}], [4,1]:[a:{x:asc 80, y:desc 30}], [4,2]:[a:{x:asc 80, y:desc 50}], [4,3]:[a:{x:asc 80, y:desc 70}]}

Figure 8.38a

vsxyx = {[1,0]:[a:{x:asc 35, y:desc 10}], [1,1]:[a:{x:asc 35, y:desc 30}], [1,2]:[a:{x:asc 35, y:desc 50}], [1,3]:[a:{x:asc 35, y:desc 70}], [2,0]:[a:{x:asc 40, y:desc 10}], [2,1]:[a:{x:asc 40, y:desc 30}], [2,2]:[a:{x:asc 40, y:desc 50}], [2,3]:[a:{x:asc 40, y:desc 70}], [3,0]:[a:{x:asc 60, y:desc 10}], [3,1]:[a:{x:asc 60, y:desc 30}], [3,2]:[a:{x:asc 60, y:desc 50}], [3,3]:[a:{x:asc 60, y:desc 70}], [4,0]:[a:{x:asc 80, y:desc 10}], [4,1]:[a:{x:asc 80, y:desc 30}], [4,2]:[a:{x:asc 80, y:desc 50}], [4,3]:[a:{x:asc 80, y:desc 70}]}

Figure 8.38b

vsxyz = {[1,0]:[a:{x:asc 20, y:desc 10, z:40}], [1,1]:[a:{x:asc 20, y:desc 30, z:40}], [1,2]:[a:{x:asc 20, y:desc 40, z:40}], [1,3]:[a:{x:asc 20, y:desc 60, z:40}], [2,0]:[a:{x:asc 40, y:desc 10, z:40}], [2,1]:[a:{x:asc 40, y:desc 30, z:40}], [2,2]:[a:{x:asc 40, y:desc 40, z:40}], [2,3]:[a:{x:asc 40, y:desc 60, z:40}], [3,0]:[a:{x:asc 60, y:desc 10, z:40}], [3,1]:[a:{x:asc 60, y:desc 30, z:40}], [3,2]:[a:{x:asc 60, y:desc 40, z:40}], [3,3]:[a:{x:asc 60, y:desc 60, z:40}], [4,0]:[a:{x:asc 80, y:desc 10, z:40}], [4,1]:[a:{x:asc 80, y:desc 30, z:40}], [4,2]:[a:{x:asc 80, y:desc 40, z:40}], [4,3]:[a:{x:asc 80, y:desc 60, z:40}]}

Figure 8.38c

\textbf{Figure A.33:} Unify two variation heads to a 2-D space with order types (Figure 8.38)
\end{verbatim}
vsxyo = {{0}: {a: {x: asc 0, y: asc 15}}, {1}: {a: {x: asc 10, y: asc 20}}, {2}: {a: {x: asc 20, y: asc 25}}, {3}: {a: {x: asc 30, y: asc 30}}, {4}: {a: {x: asc 40, y: asc 35}}, {5}: {a: {x: asc 50, y: asc 40}}, {6}: {a: {x: asc 60, y: asc 45}}}

vsxy1 = {{2}: {a: {x: asc 25, y: asc 15}}, {3}: {a: {x: asc 30, y: asc 25}}, {4}: {a: {x: asc 35, y: asc 35}}, {5}: {a: {x: asc 40, y: asc 45}}, {6}: {a: {x: asc 45, y: asc 55}}, {7}: {a: {x: asc 50, y: asc 65}}, {8}: {a: {x: asc 55, y: asc 75}}}

**Figure A.34:** Inputs of 1-D order type Cartesian unification (Figure 8.39)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

Figure 8.40a

vsxy2 = \{[0,2]:[a:asc 25, y:asc 15]], [0,3]:[a:asc 30, y:asc 25]], [0,4]:[a:asc 35, y:asc 35]], [0,5]:[a:asc 40, y:asc 45]], [0,6]:[a:asc 45, y:asc 55]], [0,7]:[a:asc 50, y:asc 65]], [0,8]:[a:asc 55, y:asc 75]], [1,2]:[a:asc 25, y:asc 20]], \ldots, [5,8]:[a:asc 55, y:asc 75]], [6,2]:[a:asc 60, y:asc 45]], [6,3]:[a:asc 60, y:asc 45]], [6,4]:[a:asc 60, y:asc 45]], [6,5]:[a:asc 60, y:asc 45]], [6,6]:[a:asc 60, y:asc 55]], [6,7]:[a:asc 60, y:asc 65]], [6,8]:[a:asc 60, y:asc 75]]\}

Figure 8.40b

vsxy2 = \{[0,2]:[a:desc 0, y:asc 15]], [0,3]:[a:desc 0, y:asc 25]], [0,4]:[a:desc 0, y:asc 35]], [0,5]:[a:desc 0, y:asc 45]], [0,6]:[a:desc 0, y:asc 55]], [0,7]:[a:desc 0, y:asc 65]], [0,8]:[a:desc 0, y:asc 75]], [1,2]:[a:desc 10, y:asc 20]], \ldots, [5,8]:[a:desc 50, y:asc 75]], [6,2]:[a:desc 25, y:asc 45]], [6,3]:[a:desc 30, y:asc 45]], [6,4]:[a:desc 35, y:asc 45]], [6,5]:[a:desc 40, y:asc 45]], [6,6]:[a:desc 45, y:asc 55]], [6,7]:[a:desc 50, y:asc 65]], [6,8]:[a:desc 55, y:asc 75]]\}

Figure 8.40c

vsxy2 = \{[0,2]:[a:desc 0, y:desc 15]], [0,3]:[a:desc 0, y:desc 15]], [0,4]:[a:desc 0, y:desc 15]], [0,5]:[a:desc 0, y:desc 15]], [0,6]:[a:desc 0, y:desc 15]], [0,7]:[a:desc 0, y:desc 15]], [0,8]:[a:desc 0, y:desc 15]], [1,2]:[a:desc 10, y:desc 15]], \ldots,[5,8]:[a:desc 50, y:desc 40]], [6,2]:[a:desc 25, y:desc 15]], [6,3]:[a:desc 30, y:desc 25]], [6,4]:[a:desc 35, y:desc 35]], [6,5]:[a:desc 40, y:desc 45]], [6,6]:[a:desc 45, y:desc 45]], [6,7]:[a:desc 50, y:desc 45]], [6,8]:[a:desc 55, y:desc 45]]\}

Figure 8.40d

Figure A.35: Cartesian unification of two 1-D spaces with order types (8.40)
\[v_{xy} = \{0: a: [x: \text{asc } 0, y: \text{asc } 0], 1: a: [x: \text{asc } 10, y: \text{asc } 0], 2: a: [x: \text{asc } 20, y: \text{asc } 0], 3: a: [x: \text{asc } 30, y: \text{asc } 30], 4: a: [x: \text{asc } 40, y: \text{asc } 40]\}\]

\[v_{yz} = \{0: a: [y: \text{asc } 0, z: \text{asc } 0], 1: a: [y: \text{asc } 10, z: \text{asc } 10], 2: a: [y: \text{asc } 20, z: \text{asc } 20], 3: a: [y: \text{asc } 30, z: \text{asc } 30], 4: a: [y: \text{asc } 40, z: \text{asc } 40]\}\]

Figure 8.41a

\[v_{xyz} = \{0,0: a: [x: \text{asc } 0, y: \text{asc } 0, z: \text{asc } 0], 0,1: a: [x: \text{asc } 0, y: \text{asc } 10, z: \text{asc } 10], 0,2: a: [x: \text{asc } 0, y: \text{asc } 20, z: \text{asc } 20], 0,3: a: [x: \text{asc } 0, y: \text{asc } 30, z: \text{asc } 30], 0,4: a: [x: \text{asc } 0, y: \text{asc } 40, z: \text{asc } 40], 1,0: a: [x: \text{asc } 10, y: \text{asc } 0, z: \text{asc } 0], \ldots, 3,4: a: [x: \text{asc } 30, y: \text{asc } 40, z: \text{asc } 40], 4,0: a: [x: \text{asc } 40, y: \text{asc } 40, z: \text{asc } 0], 4,1: a: [x: \text{asc } 40, y: \text{asc } 40, z: \text{asc } 10], 4,2: a: [x: \text{asc } 40, y: \text{asc } 40, z: \text{asc } 20], 4,3: a: [x: \text{asc } 40, y: \text{asc } 40, z: \text{asc } 30], 4,4: a: [x: \text{asc } 40, y: \text{asc } 40, z: \text{asc } 40]\}\]

Figure 8.41b

\[v_{xyz} = \{0,0: a: [x: \text{desc } 0, y: \text{desc } 0, z: \text{desc } 0], 0,1: a: [x: \text{desc } 0, y: \text{desc } 0, z: \text{desc } 10], 0,2: a: [x: \text{desc } 0, y: \text{desc } 0, z: \text{desc } 20], 0,3: a: [x: \text{desc } 0, y: \text{desc } 0, z: \text{desc } 30], 0,4: a: [x: \text{desc } 0, y: \text{desc } 0, z: \text{desc } 40], 1,0: a: [x: \text{desc } 10, y: \text{desc } 0, z: \text{desc } 0], \ldots, 3,4: a: [x: \text{desc } 30, y: \text{desc } 40, z: \text{desc } 40], 4,0: a: [x: \text{desc } 40, y: \text{desc } 0, z: \text{desc } 0], 4,1: a: [x: \text{desc } 40, y: \text{desc } 10, z: \text{desc } 10], 4,2: a: [x: \text{desc } 40, y: \text{desc } 20, z: \text{desc } 20], 4,3: a: [x: \text{desc } 40, y: \text{desc } 30, z: \text{desc } 30], 4,4: a: [x: \text{desc } 40, y: \text{desc } 40, z: \text{desc } 40]\}\]

Figure 8.41c

**Figure A.36**: Another example of order type Cartesian unification (Figure 8.41)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

vsxy = [{0}:a:{x:asc 5, y:asc 25}, {1}:a:{x:asc 17, y:asc 37}, {2}:a:{x:asc 25, y:asc 45}, {3}:a:{x:asc 37, y:asc 33}, {4}:a:{x:asc 45, y:asc 25}, {5}:a:{x:asc 33, y:asc 13}, {6}:a:{x:asc 25, y:asc 5}, {7}:a:{x:asc 13, y:asc 17}]

Figure 8.42a

vself = [{0,0}:a:{x:asc 5, y:asc 25}, {0,1}:a:{x:asc 17, y:asc 37}, {0,2}:a:{x:asc 25, y:asc 45}, {0,3}:a:{x:asc 37, y:asc 33}, {0,4}:a:{x:asc 45, y:asc 25}, {0,5}:a:{x:asc 33, y:asc 25}, {0,6}:a:{x:asc 25, y:asc 25}, {0,7}:a:{x:asc 13, y:asc 25}, {1,0}:a:{x:asc 17, y:asc 37}, ..., {6,7}:a:{x:asc 25, y:asc 17}, {7,0}:a:{x:asc 13, y:asc 25}, {7,1}:a:{x:asc 17, y:asc 37}, {7,2}:a:{x:asc 25, y:asc 45}, {7,3}:a:{x:asc 37, y:asc 33}, {7,4}:a:{x:asc 45, y:asc 25}, {7,5}:a:{x:asc 33, y:asc 17}, {7,6}:a:{x:asc 25, y:asc 17}, {7,7}:a:{x:asc 13, y:asc 17}]

Figure 8.42b

vself = [{0,0}:a:{x:desc 5, y:asc 25}, {0,1}:a:{x:desc 5, y:asc 37}, {0,2}:a:{x:desc 5, y:asc 45}, {0,3}:a:{x:desc 5, y:asc 33}, {0,4}:a:{x:desc 5, y:asc 25}, {0,5}:a:{x:desc 5, y:asc 25}, {0,6}:a:{x:desc 5, y:asc 25}, {0,7}:a:{x:desc 5, y:asc 25}, {1,0}:a:{x:desc 5, y:asc 37}, ..., {6,7}:a:{x:desc 13, y:asc 17}, {7,0}:a:{x:desc 5, y:asc 25}, {7,1}:a:{x:desc 13, y:asc 37}, {7,2}:a:{x:desc 13, y:asc 45}, {7,3}:a:{x:desc 13, y:asc 33}, {7,4}:a:{x:desc 13, y:asc 25}, {7,5}:a:{x:desc 13, y:asc 17}, {7,6}:a:{x:desc 13, y:asc 17}, {7,7}:a:{x:desc 13, y:asc 17}]

Figure 8.42c

vself = [{0,0}:a:{x:asc 5, y:desc 25}, {0,1}:a:{x:asc 17, y:desc 25}, {0,2}:a:{x:asc 25, y:desc 25}, {0,3}:a:{x:asc 37, y:desc 25}, {0,4}:a:{x:asc 45, y:desc 25}, {0,5}:a:{x:asc 33, y:desc 13}, {0,6}:a:{x:asc 25, y:desc 5}, {0,7}:a:{x:asc 13, y:desc 17}, {1,0}:a:{x:asc 17, y:desc 25}, ..., {7,0}:a:{x:asc 13, y:desc 17}, {7,1}:a:{x:asc 17, y:desc 17}, {7,2}:a:{x:asc 25, y:desc 17}, {7,3}:a:{x:asc 37, y:desc 17}, {7,4}:a:{x:asc 45, y:desc 17}, {7,5}:a:{x:asc 33, y:desc 13}, {7,6}:a:{x:asc 25, y:desc 5}, {7,7}:a:{x:asc 13, y:desc 17}]

Figure 8.42d

vself = [{0,0}:a:{x:desc 5, y:desc 25}, {0,1}:a:{x:desc 5, y:desc 25}, {0,2}:a:{x:desc 5, y:desc 25}, {0,3}:a:{x:desc 5, y:desc 25}, {0,4}:a:{x:desc 5, y:desc 25}, {0,5}:a:{x:desc 5, y:desc 25}, {0,6}:a:{x:desc 5, y:desc 5}, {0,7}:a:{x:desc 5, y:desc 17}, ..., {7,0}:a:{x:desc 5, y:desc 17}, {7,1}:a:{x:desc 13, y:desc 17}, {7,2}:a:{x:desc 13, y:desc 17}, {7,3}:a:{x:desc 13, y:desc 17}, {7,4}:a:{x:desc 13, y:desc 17}, {7,5}:a:{x:desc 13, y:desc 17}, {7,6}:a:{x:desc 13, y:desc 5}, {7,7}:a:{x:desc 13, y:desc 17}]

Figure 8.42e

Figure A.37: Order type Cartesian square unification (Figure 8.42)
\[ \text{vsxy} = \{(1,1):\{a:x:10, y:10, z:\perp\}, \ (1,2):\{a:x:10, y:20, z:\perp\}, \ (1,3):\{a:x:10, y:30, z:\perp\}, \ (1,4):\{a:x:10, y:40, z:\perp\}, \ (2,1):\{a:x:20, y:10, z:\perp\}, \ (2,2):\{a:x:20, y:20, z:\perp\}, \ (2,3):\{a:x:20, y:30, z:\perp\}, \ (2,4):\{a:x:20, y:40, z:\perp\}, \ (3,1):\{a:x:30, y:10, z:\perp\}, \ (3,2):\{a:x:30, y:20, z:\perp\}, \ (3,3):\{a:x:30, y:30, z:\perp\}, \ (3,4):\{a:x:30, y:40, z:\perp\}, \ (4,2):\{a:x:40, y:20, z:\perp\}, \ (4,3):\{a:x:40, y:30, z:\perp\}, \ (4,4):\{a:x:40, y:40, z:\perp\}, \ (5,1):\{a:x:50, y:10, z:\perp\}, \ (5,2):\{a:x:50, y:20, z:\perp\}, \ (5,3):\{a:x:50, y:30, z:\perp\}, \ (5,4):\{a:x:50, y:40, z:\perp\}, \ (6,1):\{a:x:60, y:10, z:\perp\}, \ (6,2):\{a:x:60, y:20, z:\perp\}, \ (6,3):\{a:x:60, y:30, z:\perp\}, \ (6,4):\{a:x:60, y:40, z:\perp\}\} \]

Figure 8.43a

\[ \text{vsxy1} = \{(2,1):\{a:x:20, y:10, z:\perp\}, \ (3,1):\{a:x:30, y:10, z:\perp\}, \ (3,2):\{a:x:30, y:20, z:\perp\}, \ (4,1):\{a:x:40, y:10, z:\perp\}, \ (4,2):\{a:x:40, y:20, z:\perp\}, \ (4,3):\{a:x:40, y:30, z:\perp\}, \ (4,4):\{a:x:40, y:40, z:\perp\}, \ (5,1):\{a:x:50, y:10, z:\perp\}, \ (5,2):\{a:x:50, y:20, z:\perp\}, \ (5,3):\{a:x:50, y:30, z:\perp\}, \ (5,4):\{a:x:50, y:40, z:\perp\}, \ (6,1):\{a:x:60, y:10, z:\perp\}, \ (6,2):\{a:x:60, y:20, z:\perp\}, \ (6,3):\{a:x:60, y:30, z:\perp\}, \ (6,4):\{a:x:60, y:40, z:\perp\}\} \]

Figure 8.43b

\[ \text{vsxy1} = \{(1,1):\{a:x:10, y:10, z:\perp\}, \ (2,2):\{a:x:20, y:20, z:\perp\}, \ (3,3):\{a:x:30, y:30, z:\perp\}, \ (4,4):\{a:x:40, y:40, z:\perp\}\} \]

Figure 8.43c

\[ \text{vsxy1} = \{(3,1):\{a:x:30, y:10, z:\perp\}, \ (3,2):\{a:x:30, y:20, z:\perp\}, \ (3,3):\{a:x:30, y:30, z:\perp\}, \ (3,4):\{a:x:30, y:40, z:\perp\}\} \]

Figure 8.43d

**Figure A.38**: Select variation heads from a space (Figure 8.43)
Figure A.39: Rename name of nodes and properties (Figure 8.44)
APPENDIX A. APPENDIX: VALUES OF CHAPTER EIGHT FIGURES

\[ \text{vsxyz} = \{(1,1,1):[b:x:10, y:10, z:60, c:90], a:[x:10, y:10, z:70, c:90], a:[x:10, y:10, z:20, c:10], [1,1,2]:[b:x:10, y:10, z:20, y:10, z:20, c:10], [1,2,1]:[b:x:10, y:20, z:60, c:90], a:[x:10, y:20, z:10, c:10], [1,2,2]:[b:x:10, y:20, z:70, c:90], a:[x:10, y:20, z:20, c:10], [1,3,1]:[b:x:10, y:30, z:60, c:90], a:[x:10, y:30, z:20, c:10], [1,3,2]:[b:x:10, y:30, z:70, c:90], a:[x:10, y:30, z:20, c:10], [3,3,2]:[b:x:30, y:30, z:70, c:90], a:[x:30, y:30, z:20, c:10], [4,1,1]:[b:x:40, y:10, z:60, c:90], a:[x:40, y:10, z:20, c:10], [4,2,1]:[b:x:40, y:20, z:60, c:90], a:[x:40, y:20, z:10, c:10], [4,2,2]:[b:x:40, y:20, z:70, c:90], a:[x:40, y:20, z:20, c:10], [4,3,1]:[b:x:40, y:30, z:60, c:90], a:[x:40, y:30, z:10, c:10], [4,3,2]:[b:x:40, y:30, z:70, c:90], a:[x:40, y:30, z:20, c:10]\]
Bibliography


266


Mehdi(Roham) Sheikholeslami. You can get more than you make. M.Sc., Simon Fraser University, 2009.


