



DISTRIBUTION STATEMENT R
Approved for public release
Distribution Unlimited

Algebraic Algorithm Design
and Local Search

DISSERTATION
Robert Park Graham, Jr.
Captain, USAF

AFIT/DS/ENG/96-10

19970520 214

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DTIC QUALITY INSPECTED 4

AFIT/DS/ENG/96-10

Algebraic Algorithm Design
and Local Search

DISSERTATION
Robert Park Graham, Jr.
Captain, USAF

AFIT/DS/ENG/96-10

Approved for public release; distribution unlimited

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AFIT/DS/ENG/96-10

Algebraic Algorithm Design and Local Search

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Robert Park Graham, Jr., B.S., M.S.

Captain, USAF

December, 1996

Approved for public release; distribution unlimited

Algebraic Algorithm Design and Local Search

Robert Park Graham, Jr., B.S., M.S.

Captain, USAF

Approved:

Paul D. Bailor 29 OCT 96
Paul D. Bailor, Chairman

Thomas C. Hartum 29 Oct 96
Thomas C. Hartum

James T. Moore 29 Oct 96
James T. Moore

Robert D. W. Bower 29 Oct 96
Dean's Representative

Robert A. Calico, Jr.

Robert A. Calico, Jr

Dean, Graduate School of Engineering

Acknowledgements

I would like to thank my advisor, Lieutenant Colonel Paul Bailor, for his guidance and assistance during this research effort. His integrity, knowledge, leadership and friendship made a daunting task easier to bear, and provide a model to live up to in the future. I would also like to thank my committee members, Dr. Thomas Hartrum and Lieutenant Colonel James Moore, for their advice and suggestions.

I am grateful, too, for the assistance of the fine people at the Kestrel Institute in Palo Alto, California, with particular thanks to Dr. Douglas Smith, Dr. Richard Jüllig, and Dr. Yellamraju Srinivas. Numerous visits and other contacts helped me to master the unusual tools of this trade.

I would also like to thank my fellow graduate students for encouragement, advice, distraction and comic relief, each in its proper time. How's that focus, Rick?

I also wish to thank my wife, Kathy, and my children Nathaniel, Miranda, and Alexander, who joined us in progress, for love and support throughout. It is they who bore the brunt of the day-to-day sacrifices and inconveniences. I love you all very much.

Finally, I acknowledge my greatest debt to my Lord and Saviour Jesus Christ, for all that I have and all that I may some day come to have.

Robert Park Graham, Jr.

Table of Contents

	Page
Acknowledgements	iv
List of Figures	x
List of Tables	xvii
Abstract	xviii
 I. Introduction	 1
1.1 Problem Statement	1
1.2 Related Work	3
1.2.1 Transformational Systems	6
1.2.2 KIDS	9
1.2.3 Algebraic Approaches to Software Development	15
1.2.4 SPECWARE	19
1.2.5 Algebraic Algorithm Design	22
1.2.6 Local Search	26
1.3 Overview of Document	30
 II. Notation and Terminology	 32
2.1 Specification Constructs	32
2.1.1 Specs	32
2.1.2 Morphisms	38
2.1.3 Diagrams and Colimits	43
2.1.4 A Final Example	47
2.2 Refinement Constructs	48
2.2.1 Interpretations	48
2.2.2 Interpretation Morphisms	53
2.2.3 Diagram Refinement	53

	Page
III. Characterizing Local Search Algorithms	57
3.1 Overall Strategy	59
3.2 Finding an Initial Solution	60
3.3 Search Strategy	61
3.3.1 Neighborhood Structure	61
3.3.2 Move Selection Rules	63
3.3.3 Stopping Criteria	65
3.4 Examples of Local Search Algorithms	67
3.5 Problem Relaxation	77
3.6 Conclusion	78
IV. Algebraic Algorithm Design	79
4.1 Specifying Problems	79
4.1.1 A Canonical Form for Problem Specification	79
4.1.2 A Canonical Form for Finding All Solutions to a Problem	81
4.1.3 Canonical Forms for Global Optimization Problems	83
4.1.4 A Canonical Form for Constraint Satisfaction Problems	85
4.1.5 The Value of Canonical Forms	87
4.2 Specifying Solutions	89
4.3 Algorithm Design	90
4.4 Algorithm Refinement	95
4.5 Conclusion	98
V. Completing Interpretation Morphisms	99
5.1 Syntactic Methods for Completing Interpretation Morphisms	101
5.2 Connections between Specifications	105
5.2.1 A Gentle Introduction to Connections	105
5.2.2 Formal Connection Theory	110

	Page
5.2.3 Polarity Analysis	114
5.2.4 Connections in Practice	122
5.2.5 Example Connection	128
5.3 Conclusion	142
VI. Formalizing Basic Local Search	145
6.1 Neighborhood Structure	147
6.2 Neighborhood Matching Tactic	157
6.2.1 Applying Connections to <i>Neighborhood</i>	158
6.2.2 Lowry's Design Tactic for Local Search	162
6.2.3 A Revised Tactic for Designing Specialized Neighborhoods	167
6.2.4 Example	174
6.3 Program Schemes for Basic Local Search	187
6.3.1 Hill Climbing	190
6.3.2 Hill Climbing with a Single Trial	195
6.3.3 Steepest Ascent Hill Climbing	196
6.3.4 Hill Climbing with Neutral Moves	201
6.3.5 Simulated Annealing	206
VII. Advanced Local Search Techniques	222
7.1 Tabu Search	223
7.1.1 Principles of Tabu Search	224
7.1.2 Variations and Examples of Tabu Strategy	239
7.1.3 Aspiration Criteria	253
7.1.4 Program Scheme for Tabu Search	264
7.1.5 Example: Tabu Search for the Graph Partitioning Problem	276
7.2 The Kernighan-Lin Heuristic	287
7.2.1 Program Scheme for Kernighan-Lin	290

	Page
7.2.2 Experiments with Neighborhoods	295
7.2.3 Experiments with Boolean Satisfaction	304
VIII. Conclusions	307
8.1 Contributions	307
8.2 Discussion	311
8.3 Future Work	316
Appendix A. Library of Neighborhood Structures	320
A.1 k -Subset-1-Exchange	322
A.2 k -Subset-2-Exchange	324
A.3 Free-Subset	326
A.4 Independent-Set	328
A.5 2-Partition-1-Exchange	331
A.6 Map-Swap-Domain	332
A.7 Map-Swap-Range	334
A.8 Map-Set-Var-to-Val	336
A.9 Array-Swap-Index	337
A.10 Array-Swap-Adjacent-Index	338
A.11 Array-Insert-Element	339
A.12 Ring-Swap-Adjacent-Index	340
A.13 Ring-Reverse-Subarray	341
A.14 Ring-Move-Subarray	343
Appendix B. A Proof About Classification Diagrams	345
Appendix C. Lowry's Theory of Basic Neighborhoods	351
C.1 Basic Neighborhood Theory	351
C.2 Evaluation of Basic Neighborhood Theory	354
C.3 Conclusions	358

	Page
Bibliography	360
Vita	365

List of Figures

Figure	Page
1. Software Development as an Engineering Process	4
2. Specification of the Global Search Algorithm Theory	23
3. Specification of Basic Problem Theory	25
4. Ladder Diagram for Classification Approach to Design	27
5. Abstract Local Search Theory	29
6. A Basic Spec for a Generic Total Order	33
7. A Basic Spec for the Natural Numbers	34
8. Basic Spec for Finite Maps	37
9. Import Example	38
10. Specs for Collection Classes	41
11. Extending <i>BasicSet</i> to <i>Set</i>	43
12. Properties of Binary Relations	46
13. Spec for a Set of Natural Numbers	47
14. Diagram for Propositional Logic Theory	48
15. Building a Theory of Propositional Logic	49
16. Refining Sets to Sequences	51
17. Composition of Interpretations	52
18. Parallel Composition of Interpretation Scheme Morphisms	54
19. Example of Diagram Refinement	55
20. Diagram Refinement	56
21. Simplex Algorithm for Linear Programs	69
22. GSAT Algorithm for Boolean Satisfiability	70
23. The Kernighan-Lin Algorithm for Graph Partitioning	72
24. Generic Simulated Annealing Algorithm	73
25. Generic Tabu Search Algorithm	75
26. A Simple, Generic Genetic Algorithm	76

Figure	Page
27. Boolean Satisfaction as a Problem in Propositional Logic	80
28. Constructing a Spec for a Set of Solutions	82
29. Problem Specification for Finding All Solutions	82
30. Constructing a Specification for All-BoolSat Problem	83
31. Canonical Form for Global Optimization Problems	84
32. Constraint Satisfaction Domain Theory	86
33. Specification of Constraint Satisfaction Problem	87
34. Boolean Satisfiability Expressed as Constraint Satisfaction	88
35. Canonical Form for a Solution to a Problem	89
36. Problem Classification	91
37. Boolean Satisfiability Classified as Constraint Satisfaction	92
38. Details of Boolean Satisfiability Classification	92
39. Alternative Classification Diagram for <i>BoolSat</i>	94
40. An Algebraic Program Scheme	96
41. Instantiating a Program Scheme	96
42. General Form of Algorithm Refinement	98
43. Completing an Interpretation Morphism Given the Source	100
44. Completing an Interpretation Morphism Given the Target	100
45. Extending an Interpretation Along a Definitional Extension	101
46. Identity Completion of an Interpretation Morphism	103
47. Composing a Morphism with an Interpretation	103
48. The Spec <i>Program</i>	106
49. Venn Diagram for <i>Program Spec</i>	106
50. Venn Diagram for Matching an Operator	107
51. Completing an Interpretation Morphism using Connections	111
52. Using a Shape Morphism to Combine Specs	114
53. Building a Domain Theory for the <i>K</i> -Queens Problem	129

Figure	Page
54. Specs for a K -Queens Domain Theory	130
55. Specification of the K -Queens Problem	131
56. The Global Search Algorithm Theory	132
57. Domain Theory for Enumerating Maps	133
58. Problem Specification for Enumerating Maps	134
59. Global Search Applied to Map Enumeration	136
60. Global Search Applied to Map Enumeration, Cont.	137
61. A <i>GlobalSearch</i> -Connection for K -Queens, Steps 1–3	138
62. A <i>GlobalSearch</i> -Connection for K -Queens, Step 3 Details	139
63. A <i>GlobalSearch</i> -Connection for K -Queens, Step 4	141
64. A <i>GlobalSearch</i> -Connection for K -Queens, Step 5	143
65. Local Search Theory	146
66. Problem Specification for Local Optimization	146
67. Exact Local Search Solves Global Optimization Problems	147
68. Specification of a Theory of Neighborhood Structure	148
69. Extending Local Optimization to a Feasible Neighborhood	151
70. Specification of k -Subset Solution Space	155
71. k -Subset-1-Exchange Neighborhood	156
72. Lowry's Program Scheme for Hill Climbing	164
73. Overview of Neighborhood Matching Tactic	171
74. Domain Theory for Graph Partitioning	176
75. Domain Theory for Graph Partitioning, Cont.	177
76. Problem Specification for Graph Partitioning	178
77. Neighborhood Tactic for Graph Partitioning, Steps 1–3	180
78. Combining Graph Partitioning and k -Subset-1-Exchange	181
79. Neighborhood Tactic for Graph Partitioning, Step 4	182
80. Neighborhood Tactic for Graph Partitioning, Step 5	184

Figure	Page
81. Neighborhood Tactic for Graph Partitioning, Step 6	185
82. Cleaning Up the Neighborhood for Graph Partitioning	186
83. Local Search Specification for Graph Partitioning	187
84. Domain Theory for Strict Hill Climbing	191
85. Mediator Spec for Strict Hill Climbing	193
86. Program Scheme for Local Optimization via Strict Hill Climbing	194
87. Refining <i>HillClimb</i> for Single-Trial Hill Climbing	197
88. Program Scheme for Single-Trial Hill Climbing	198
89. Simplified Program Scheme for Single-Trial Hill Climbing	199
90. Refining <i>HillClimbing</i> to <i>HillClimbing1</i>	200
91. Hill Climbing Refined to Steepest Ascent Hill Climbing	202
92. Program Scheme for Hill Climbing with Neutral Moves	203
93. Mediator Spec for Hill Climbing with Neutral Moves	204
94. Domain Theory for Abstract Simulated Annealing	208
95. Domain Theory for Abstract Simulated Annealing, Cont.	209
96. Mediator Spec for Abstract Simulated Annealing	212
97. Program Scheme for Abstract Simulated Annealing	213
98. Domain Theory for Run-Length-Based Simulated Annealing	215
99. Domain Theory for Run-Length-Based Simulated Annealing, Cont.	216
100. Domain Theory for Run-Length-Based Simulated Annealing, Cont.	217
101. Refinement of Abstract Simulated Annealing to Run-Length-Based	220
102. Hierarchy of Specs for Tabu Rules	233
103. Inverse Move Tabu Strategy	234
104. Repeat Move Tabu Strategy	235
105. Repeat and Inverse Move Combined Tabu Strategy	236
106. Lock-In Strategy Applied to k -Subset-1-Exchange Neighborhood	238
107. Lock-In Strategy Applied to k -Subset-1-Exchange Neighborhood, Cont.	239

Figure	Page
108. Inverse Move Strategy Instantiated for k -Subset-1-Exchange Neighborhood	240
109. Incorporating <i>TabuRule</i> into <i>LocalSearch</i>	241
110. Tabu Rules for k -Subset-1-Exchange Derived from the Lock-Out Rule	244
111. Tabu Rules for k -Subset-1-Exchange Derived from the Inverse Move Rule	246
112. Lock-Out and Lock-In Rules for Array-Swap-Index Neighborhood	247
113. Tabu Rules for Array-Swap-Index Derived from the Lock-Out and Lock-In Rules	249
114. Tabu Rules for Array-Swap-Index Derived from the Inverse Move Rule	250
115. Lock-Out and Lock-In Rules for Map-Set-Var-to-Val Neighborhood	252
116. Basic Elements of Aspiration Criteria	255
117. Aspiration by Global Objective	257
118. Forward-Looking, Cost-Level Aspiration With Cost as Attribute	259
119. Backward-Looking, Cost-Level Aspiration With Cost as Attribute	260
120. Forward-Looking, Cost-Level Aspiration With Tabu Attribute	262
121. Backward-Looking, Cost-Level Aspiration with Tabu Attribute	263
122. Aspiration By Search Direction	265
123. Spec for Sequences	266
124. Domain Theory for Tabu Search with Aspiration	267
125. Domain Theory for Tabu Search with Aspiration, Cont.	268
126. Mediator Spec for Tabu Search with Aspiration	271
127. Mediator Spec for Tabu Search with Aspiration, Cont.	272
128. Mediator Spec for Tabu Search with Aspiration, Cont.	273
129. Program Scheme for Tabu Search with Aspiration	275
130. Adapting Lock-In for k -Subset-1-Exchange to Graph Partitioning, Steps 1-3	279
131. Combining Lock-In for k -Subset-1-Exchange with Graph Partitioning	280
132. Adapting Lock-In for k -Subset-1-Exchange to Graph Partitioning, Step 4	282
133. Adapting Lock-In for k -Subset-1-Exchange to Graph Partitioning, Step 5	284
134. Tabu Search for Graph Partitioning	287

Figure	Page
135. Instantiating Aspiration by Global Objective for Graph Partitioning	288
136. Final Instantiation of Tabu Search for Graph Partitioning	289
137. Domain Theory for Kernighan-Lin Search	291
138. Program Scheme for Kernighan-Lin Search	292
139. Mediator Spec for Kernighan-Lin Search	293
140. Mediator Spec for Kernighan-Lin Search, Cont.	294
141. Maximum Run Length for k -Subset-1-Exchange with Lock-Out, $k = 3$	299
142. Maximum Run Length for k -Subset-1-Exchange with Lock-Out, $k = n/2$	299
143. Mean Run Length for Array Swap Index with Lock-Out	302
144. Mean Run Length for Array Swap Index with Free-Conjunctive Lock-Out	302
145. Mean Run Length for Array Swap Index with Single-Attribute Lock-Out	303
146. Mean Run Length for Array Swap Index with Disjunctive Lock-Out	303
147. k -Subset-1-Exchange	322
148. k -Subset-2-Exchange	324
149. Free-Subset	326
150. Independent-Set	328
151. 2-Partition-1-Exchange	331
152. Map-Swap-Domain	333
153. Map-Swap-Range	335
154. Map-Set-Var-to-Val	336
155. Array-Swap-Index	337
156. Array-Swap-Adjacent-Index	338
157. Array-Insert-Element	339
158. Ring-Swap-Adjacent-Index	340
159. Ring-Reverse-Subarray	341
160. Ring-Move-Subarray	343
161. Ring-Move-Subarray Operation	344

Figure	Page
162. Classification Diagram	346
163. Top, Right and Center Interpretations of Classification Diagram	347
164. Composition of Top and Right Interpretations	347
165. Testing Equality of Diagonal and Composed Interpretations	348
166. Lemma Diagram	349
167. Mapping from Abstract Local Search Theory to Basic Neighborhood Theory	353
168. Binary Tree Rotation	358

List of Tables

Table		Page
1.	Observed Run Lengths for k -Subset-1-Exchange with Free Conjunction	298
2.	Comparing Three Local Search Algorithms for Boolean Satisfiability	305

Abstract

Formal, mathematically-based techniques promise to play an expanding role in the development and maintenance of the software on which our technological society depends. Algebraic techniques have been applied successfully to algorithm synthesis by the use of *algorithm theories* and *design tactics*, an approach pioneered in the Kestrel Interactive Development System (KIDS). An algorithm theory formally characterizes the essential components of a family of algorithms. A design tactic is a specialized procedure for recognizing in a problem specification the structures identified in an algorithm theory and then synthesizing a program. Design tactics are hard to write, however, and much of the knowledge they use is encoded procedurally in idiosyncratic ways. Algebraic methods promise a way to represent algorithm design knowledge declaratively and uniformly.

We describe a general method for performing algorithm design that is more purely algebraic than that of KIDS. This method is then applied to *local search*. Local search is a large and diverse class of algorithms applicable to a wide range of problems; it is both intrinsically important and representative of algorithm design as a whole. A general theory of local search is formalized to describe the basic properties common to all local search algorithms, and applied to several variants of hill climbing and simulated annealing. The general theory is then specialized to describe some more advanced local search techniques, namely tabu search and the Kernighan-Lin heuristic.

KEYWORDS: Software synthesis, algorithm design, algebraic methods, category theory, local search, tabu search, simulated annealing, Kernighan-Lin heuristic.

Algebraic Algorithm Design and Local Search

I. Introduction

1.1 Problem Statement

Computers and their associated software are increasingly pervasive and critical to modern technological society. The need for ever greater amounts of increasingly complex software that is reliable and economical to produce is well documented. Formal, mathematically-based techniques promise to play an expanding role in the development and maintenance of this software. Algorithms ultimately lie at the heart of software, so formal methods for algorithm design are a critical part of this effort. Local search is a large and diverse class of algorithms applicable to a wide range of problems, and so is both intrinsically important and representative of algorithm design as a whole.

Algebraic techniques have been applied successfully to algorithm synthesis by the use of *algorithm theories* and *design tactics*, an approach pioneered in the Kestrel Interactive Development System (KIDS) (69, 73). An algorithm theory is an algebraic specification that formally characterizes the essential components of a family of algorithms, such as global search or divide-and-conquer. A design tactic is a specialized procedure for recognizing in a problem specification the structures identified in the algorithm theory and then synthesizing a program. Some problems with design tactics, however, are that they are hard to write, and they encode knowledge procedurally in idiosyncratic ways, making it hard to modify or expand the knowledge base.

Local search is a technique for solving optimization problems. Such problems are defined by a set of feasible solutions and a cost or objective function that assigns a numerical value to each solution. An optimization problem is solved by finding a solution whose cost is at least as good as the cost of every other solution. Local search algorithms can in some cases find solutions that are known to be optimal, or within a certain factor of an optimal solution. For many optimization

problems, however, the known algorithms for solving them exactly are of exponential complexity, making them impractical for solving large instances. Local search is commonly used in these cases to generate good solutions even though no guarantees on solution quality are provided, and as a cheap way to improve solutions generated by other heuristic means.

The goal of this investigation was to define a formal model of local search algorithms that supports their semi-automated synthesis. Our approach was to follow KIDS in spirit, but to adopt a pure algebraic formalism, supported by Kestrel's SPECWARE environment (79), that would provide a uniform and declarative representation for algorithmic design knowledge. To this end, a general method for performing algorithm design was developed that is more purely algebraic than that of KIDS. This method was then applied to local search. A general theory of local search was formalized to describe the basic properties common to all local search algorithms and then was specialized to some of the more advanced techniques currently in use. Examples applying this theory to particular problems were developed throughout. Specific contributions of this research include:

- An informal characterization of local search algorithms, their features and variants
- A more complete algebraic theory of algorithm design than is provided in the KIDS work
- An algebraic theory of *connections between specifications*, a technique used in some KIDS tactics for reusing existing algorithmic knowledge, and a clearer explanation of the connection mechanism
- A detailed analysis of neighborhood structures and their properties
- A procedure for adapting neighborhoods to particular problems while maintaining desirable properties
- Formalization of two basic local search techniques, hill climbing and simulated annealing

- Specialization of the general local search theory to support two advanced local search techniques, tabu search and the Kernighan-Lin heuristic

1.2 Related Work

The results achieved for algorithm design and the synthesis of local search algorithms are an application of formal methods to software engineering and falls in the larger context of knowledge-based software engineering (KBSE). Software engineering broadly is the use of a disciplined approach to the development and maintenance of computer software. The goals of software engineering are to improve the productivity of those who develop and maintain software, improve the reliability and efficiency of the software they produce, and make the entire process more predictable and less expensive. More specifically, software engineering attempts to adapt traditional engineering design methodologies to the design of software artifacts. Most software development today is *ad hoc*: there is little in the way of accepted practice to guide the development of solutions to problems. Innovation is highly valued, routine design disparaged and avoided. Traditional engineering practice is quite different. Here problems are formulated and classified in terms of known problem classes and solution techniques. Innovation and creativity are primarily expressed in terms of finding new arrangements of existing components rather than resolving old problems in idiosyncratic ways. Figure 1 provides one view of software development as an engineering process (4).

Engineering, including software engineering, is a knowledge-intensive activity. Knowledge about software architectures, algorithms, data structures, time and space tradeoffs, programming languages, code optimization and machine architectures must all be integrated and brought to bear on a problem to solve it efficiently and effectively. The resulting artifact should be correct, efficient and easy to maintain, which implies that its structure must be comprehensible and its complexity controlled. This is a stringent set of requirements, and one of the causes of the "software crisis" has been our inability to accumulate and reuse the required knowledge. KBSE can be broadly

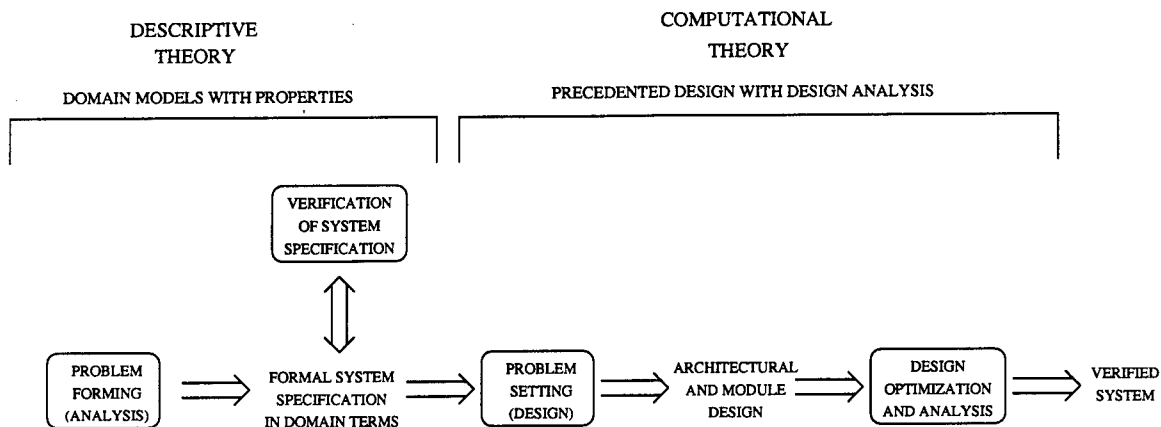


Figure 1. Software Development as an Engineering Process

described as the attempt to identify and formalize software design knowledge and embed it in a variety of automated aids. Formal methods are one approach to this problem that is being tried.

Formal methods are the tools of a mathematical approach to representing and manipulating knowledge. Classical applications are in artificial intelligence (AI) and the theory of computation, among others. Knowledge is represented in a number of languages whose syntax and semantics are rigorously defined, including lambda calculus, production rules, and various logical systems (e.g., propositional logic, Horn clauses, and first-order, higher-order and modal logics). Knowledge can be structured as rules, frames, plans, scripts, schemas, nets, algebras and in other ways. Automated theorem proving technology is often employed to automate or assist the application of formal methods to particular problems.

Formal methods have many applications to software engineering. Two broad areas of interest are specification acquisition and software synthesis. Formal specification languages provide a means to describe a system abstractly and unambiguously. Properties of specifications can be formally verified. Some specification languages can be executed to provide an early prototype of the system being built. Z is a set-theoretic language where a system is specified as a set of interrelated *schemas* that define types and operations and their properties in first-order logic; it is not executable, however (60). Formal specifications can be used to augment popular informal techniques such as object-

oriented analysis and design (37). Approaches to formal specification based on modern abstract algebra will be discussed later.

Specifications can be implemented by informal means, or a program can be synthesized by formal means. One long-term goal of KBSE is to automate the software synthesis task to the point where it is feasible to perform all maintenance at the level of the formal specification and re-synthesize a program rather than directly modifying code written in a traditional programming language, where the purpose and structure of the program is obscured by a mass of optimizations, error checking, and other low-level detail. Formal approaches to software synthesis fall broadly into two classes: deductive and transformational.

Deductive program synthesis structures the synthesis task as a theorem-proving task. Manna and Waldinger (55) describe one such method. A problem is formally specified by an input relation that describes the properties of valid inputs and an output relation that describes the properties of correct solutions. A theorem is then formulated that a function exists that implements the specified behavior. In the process of proving this theorem, a program is created that is correct by construction. In order for the proof to be sufficiently constructive, a specialized proof system called deductive tableaux is employed. The method has a number of drawbacks, primarily due to its weak knowledge representation. The method has no representation for what is called coarse-grained knowledge, such as knowledge about algorithms or data structures. The various inference rules and proof steps represent knowledge comparable to individual programming language constructs. Therefore, each proof step accomplishes little and proofs are long. They are also unstructured and hard to understand. The method is difficult to automate because the search space is too unconstrained and so blows up quickly, and applying it manually requires substantial mathematical sophistication and experience. Other deductive methods such as the predicate transform techniques proposed by Dijkstra and Gries, Floyd, Hoare, and others suffer similar drawbacks and additionally are targeted to particular programming languages (4).

1.2.1 Transformational Systems. Transformational software synthesis has had more success than the deductive approach (51). A transformation is a rule that describes how one software element can be transformed into another when certain conditions hold. Transformations that are proven valid are called correctness preserving. Reliance on correctness-preserving transformations provides automatic, implicit verification of the resulting program. Automated theorem proving is often used, but in a supporting role to detect and verify the application of transformations rather than as the primary inference mechanism as in the deductive approach.

Some sets of transformation rules are known to be complete in the sense that arbitrary program transformations can be carried out by composing them appropriately. For example, the transformations *fold* and *unfold* are such a set. *Folding* refers to replacing an expression with a call to a function with that expression as its definition. *Unfolding* is the converse operation that replaces a function call with its definition. Small, complete sets of transformations are elegant and can be used to prove important theoretical results about computation, but are impractical to use for program synthesis and analysis tasks. Like the deductive approach, they lead to long, convoluted derivations even of simple programs and fail to exploit all of the kinds of knowledge that a software engineer needs to have.

Reliance on small, fixed sets of transformations, even complete sets, fails to exploit the full power of the transformational approach. Its key strength is precisely its ability to integrate coarse and fine-grained knowledge across the continuum. For example, a low-level transformation might state that the expressions $x + 0$ and x are always equivalent. A higher-level transformation might describe how arrays can be used to implement the stack abstract data type. Transformations can represent algorithmic or architectural knowledge just as easily.

Applications of transformations are not limited to synthesis but include a variety of tasks. During synthesis, abstract descriptions are refined into lower-level implementations. During analysis an implementation is recognized as an instance of a higher-level abstraction. Equivalences like

$x + 0 = x$ can be used as simplifications to optimize program code, or used to "jitter" a program fragment into the form needed by some other transformation.

Because of the simple, uniform structure of transformations, they are relatively easy to incorporate into automated or semi-automated tools. The Refine¹ language provides direct support for using transformations to manipulate abstract syntax trees and other data structures (61). Automation relieves the user of having to deal with a lot of low-level detail and helps insure that correctness-preserving transformations are in fact carried out correctly.

The primary difficulty in creating fully automated tools is the problem of search control. We have seen that relying on a small number of very general and therefore fine-grained rules leads to a combinatorial explosion because derivations are long. Coarse-grained rules encode larger chunks of knowledge, but the number of such rules is itself combinatorial. It is difficult to capture the right abstractions that are higher-level but still general, which is what design knowledge represents. Available evidence suggests that a library on the order of several thousand design-level concepts would be needed for a comprehensive understanding of programming (83). This estimate omits still higher levels of abstraction such as architectural design. Known methods for controlling a search through the space of possible derivations are still relatively weak, so the majority of existing program synthesis systems are still only semi-automated and rely on the user to make decisions about which transformations to apply; the rest are very restrictive in the domains they address, the kinds of programs they can synthesize, or both.

Many transformation-based systems have been developed, though not all of them are formal. A few will be described briefly to indicate the variety of uses to which the transformational approach has been applied.

1. ELF (65) automatically synthesizes wire-routing software for VLSI and printed circuit board layout by combining knowledge about routing algorithms, data structures, device fabrica-

¹Refine is a trademark of Reasoning Systems, Inc

tion characteristics, application characteristics, and the constraints and interactions among these components. Several routers generated by ELF have been competitive with commercial products.

2. Medusa (57) accepts a functional specification of a program and a performance constraint in the form of asymptotic time complexity and successively refines the specification into a correct program that satisfies the constraint. Dependencies among the subtasks of a refinement can lead to combinatorial explosion, so Medusa restricts the kinds of dependencies it allows and also reasons directly about the remaining dependencies to avoid unnecessary and duplicated effort.
3. The Requirements Apprentice (62) is an analysis tool that addresses the problem of formalizing a set of requirements into a specification. The apprentice has a library of common system types and their requirements, related hierarchically, which provides a high-level vocabulary for communicating with the analyst and a basis for anticipating unstated requirements, filling in missing detail, and detecting contradictions and some classes of missing information.
4. DTRE (6) is a transformational system that supports the formal specification of abstract data types and their verified refinement into efficient implementations. The implementation of DTRE described in the work cited contains a library of refinements that can be used to implement set-theoretic types such as sets, maps, sequences and tuples in terms of arrays, lists, bit-vectors and so forth. Refinement selection is semi-automatic and based in part on the data operations used and their relative frequency of occurrence.
5. Lubars (52) has created a series of systems that implement his knowledge-based refinement paradigm in which user requirements are used to select an abstract design schema from a library and customize it. Each schema has an associated set of issues to resolve. Resolution of these issues by the user triggers transformations that carry out the customization. Constraints between issues may trigger transformations automatically.

6. KBEmacs (83) is a knowledge-based editor that can perform high-level program manipulations automatically. It uses a library of highly parameterized program schemas or *clichés* to define a vocabulary of intermediate-level programming concepts. In addition to the usual editor commands, a small set of "knowledge" commands allow the user to retrieve and instantiate clichés, provide an argument for a parameter or determine whether two computations can be shared. The knowledge incorporated into KBEmacs is still relatively low level, but Waters reports productivity gains of factors from 3 to 10 over manual coding.
7. Rich's plan calculus (63) includes four formal relations among plans. One is called an overlay and can be used as a transformation on plans.

One transformation system deserves special consideration, because it points the way to especially promising approaches to general system development and because it provides much of the foundation on which the current work on algorithm design and local search is built. This system is KIDS.

1.2.2 KIDS. The Kestrel Interactive Development System (KIDS) (69) brings together in one system a lot of earlier ideas about refinement via transformations along with significant new ones, and integrates them into a graphical, user-friendly prototype environment that strongly suggests a truly practical method for doing software synthesis. KIDS uses a language called Regroup, which is an extension of the Refine language, for describing both domain theories and programs. KIDS provides a set of correctness-preserving refinement operators that the user applies to a program fragment by selecting it with a mouse. KIDS computes the result of the transformation and updates the program. Derivations can be stored for future replay, and derivation steps can be undone so alternate derivations can be explored.

The Regroup language allows a function to be specified using the following syntax:

function $f(x : D \mid I(x))$ returns $(z : R \mid O(x, z))$

where D and R are the input and output types, respectively, I is the input condition (or precondition)

tion) and O is the output condition (or post-condition). An optional function body following the specification describes how the computation is carried out. As long as a body is provided, function definitions can be compiled and executed. If a body is provided, parts of the specification may be omitted, namely I and O . If a complete specification and a body are provided, KIDS does not verify that they are consistent.

The main user input to KIDS is a set of domain theories written in Regroup. A domain theory consists of a number of functions for which the user provides specifications, bodies or both, and a number of *theory laws* describing properties of these functions. The theory laws are used by an integrated theorem prover, called Rainbow, to simplify expressions and otherwise reason about them. Rainbow has no direct access to the specifications or bodies of the functions of a theory; it knows only the knowledge encoded in the laws. KIDS contains several pre-compiled theories describing the built-in types and functions of Regroup and their interactions. The domain theory for sets, for example, includes the law $size(\{\}) = 0$, where $\{\}$ represents the empty set.

Most of the refinement operators in KIDS are program optimizations that are applied to function bodies; I will describe a few of these. There are two simplification operators. One is called context-independent simplification, which exhaustively applies a set of rewrite rules (a type of theory law) to replace expressions with equivalent but "simpler" ones. The other kind of simplification is called context-dependent. This form uses information about the context of an expression when trying to simplify it. The context of an expression is any predicate known to be true whenever the expression is evaluated, and includes the input condition of the surrounding function, the conditions associated with any surrounding if or while statements, sibling predicates in a conjunct, and so forth. The Rainbow theorem prover uses this rich set of assumptions to simplify expressions by removing redundancy.

The unfold operator replaces a function call with a copy of its body, instantiating the parameters with the arguments of the call. Context-dependent simplification can then be used to simplify

the in-line body, taking advantage of the particular form of the arguments and the surrounding context. This process of unfolding and then simplifying accomplishes what is sometimes called partial evaluation.

The finite differencing operator replaces a computation with a stored data object. This increases program efficiency whenever the object can be incrementally updated more cheaply than it can be recomputed. The KIDS implementation decomposes finite differencing into two steps: abstraction and simplification. The user selects an expression $E(x)$ in the body of function f and provides a name for a variable, say c , to represent it. The operator then abstracts f with respect to the expression by adding c as a new parameter and $c = E(x)$ as a new input condition. All calls to f are updated to supply the new argument such that the new input condition is satisfied. These calls and the entire body of f are then simplified, which replaces occurrences of $E(x)$ with simplified or incremental forms.

KIDS provides several other operators for optimizing a program, but the most original contributions of KIDS are its *design tactics*, which are operators that perform automated algorithm design (73). A design tactic takes a function specification that lacks a body and derives an algorithm of a particular style that correctly implements it. Each tactic is based on a formal model of an algorithm class that abstractly defines the components (e.g., data types and operations) of algorithms in the class and the constraints that these components must satisfy. The tactic carries out a sequence of steps that derive definitions for these components in terms of the domain theory of the function, then uses these definitions to instantiate a program schema that describes how the components are combined into a concrete algorithm. For example, KIDS has a design tactic for designing *divide and conquer* algorithms (68). Informally, this class of algorithms decomposes a problem instance into smaller problem instances until a set of primitive problems is obtained, then solves each of these directly and composes the solutions to form a solution to the original problem. The formal model for this class of algorithms defines *Compose*, *Decompose*, *Primitive*,

and *DirectlySolve* as abstract operations and describes how they relate to each other. One of the constraints is the following condition that must hold between *Compose* and *Decompose*:

$$\begin{aligned} \forall(x_0, x_1, x_2 : D) \forall(z_0, z_1, z_2 : R) (I(x_0) \wedge Decompose(x_0, x_1, x_2) \\ \wedge O(x_1, z_1) \wedge O(x_2, z_2) \wedge Compose(z_0, z_1, z_2) \Rightarrow O(x_0, z_0)) \end{aligned} \quad (1)$$

where D , R , I and O represent a generic function specification, as before. This condition states that if x_0 is a valid input that decomposes into x_1 and x_2 , and if these are respectively solved by z_1 and z_2 that can be composed into z_0 , then z_0 must be a solution for x_0 .

The design tactic based on this model queries the user for a problem-specific definition of either the *Compose* or *Decompose* operation, and from that is able to derive more detailed specifications and/or bodies for all of the other operations automatically by using a theorem prover and conditions like the one above that the operations must satisfy. The user is then given a choice of program schemas that combine the derived operations in different ways, for example by using parallel vs. sequential or iterative vs. recursive control strategies. The tactic thus provides a clear separation between independent issues in design. Sorting is a problem that is frequently solved with a divide and conquer algorithm. Different choices for *Compose* and *Decompose* lead to different algorithms: quick sort, heap sort, shell sort and so on.

When a tactic derives only a specification for a function, the user must then apply another (or the same) design tactic recursively to derive an algorithm for it. This is not a flaw in the approach but rather defines a top-down approach to algorithm design that is very powerful as it defers some issues and gives the designer complete freedom in implementing subproblems. Top-down design is of course widely advocated in numerous manual design methodologies.

In addition to divide and conquer, KIDS has tactics for problem reduction (which includes dynamic programming) (70) and global search (which includes backtrack and branch-and-bound)

(69). The tactic for global search introduces some new concepts and techniques that divide and conquer does not use.

The informal idea behind global search is to search a space of possible solutions systematically and exhaustively. The search makes use of *subspace descriptors* to represent subsets of possible solutions. It begins with an initial subspace containing all possible solutions and successively splits subspaces into smaller subspaces and extracts solutions from them until no subspaces remain to be split. To find a schedule for a set of jobs that satisfies some set of constraints, for example, a partial schedule serves as a subspace descriptor by denoting all possible ways of extending it to a complete schedule. The empty schedule serves as the initial subspace. Operations in the formal theory of global search include *Split*, *Extract*, $\hat{r}_0$, which computes the initial subspace descriptor, and the usual *D*, *R*, *I* and *O*.

The key step in synthesizing a global search algorithm is finding a suitable subspace descriptor. The tactic uses a library of *subspace generators* that define subspace descriptors for common output data types, such as sets, sequences and maps. Subspace generators provide definitions for the operations of global search and can be considered specialized models for global search over certain classes of solution space. The tactic extracts one of these models from the library and further specializes it to a particular function specification. In order for a subspace generator *A* to be applicable to a function specification *B*, the following condition must hold:

$$R_A = R_B \wedge \forall(x : D_B) \exists(y : D_A) \forall(z : R_B) (I_B(x) \wedge O_B(x, z) \Rightarrow O_A(y, z)) \quad (2)$$

In the process of proving this condition, the tactic derives the information necessary to define global search operations specialized for *B*. As with the divide and conquer tactic, these definitions are used to instantiate a program schema.

The efficiency of global search is greatly enhanced by the use of *filters*. A filter is a test that detects when a subspace does not contain any solutions. For example, if job *I* is required to

precede job J , then any partial schedule with J before I can be deleted or *pruned* from the search. A separate inference step before a program scheme is selected derives a set of filters from which the user may select one or more. These filters are incorporated into the final algorithm as well.

Global search and the use of filters have been extended in more recent work to the notion of *constraint propagation*. Constraint propagation formalizes the idea that the parts of a solution can interact, so that the choices made in a partial solution can constrain the remaining choices. A filter is a special case where the choices made leave no way to complete a solution. More generally, after each split a subspace can be pared down by making explicit the consequences of the choice implicit in the split. For example, if job I is required to follow job J immediately, then any time I is scheduled, J can be automatically scheduled as well. Automated methods for synthesizing highly efficient code for propagating constraints for problems in the transportation scheduling domain has produced schedulers that are orders of magnitude faster than other deployed scheduling programs (74, 75).

For all their power, design tactics as currently implemented in KIDS suffer a number of drawbacks. For one, they are complex and therefore difficult to write and to verify. The inference steps have to be carefully constructed to insure correctness of the generated algorithm and are not necessarily intuitively obvious. Program schemes must also be verified. These disadvantages are mitigated by the fact that each tactic only needs to be verified once and can be used many times.

Second, although each tactic has been used to synthesize algorithms for a wide range of domains, they are often brittle or hard to use: they fail on problems that seem very similar to ones on which they succeed, and can be quite demanding about the form of the specification to which they are applied or the theory laws defined for the domain. The number of general approaches to algorithm design is relatively small, but the number of tactics that would be needed to encompass most or all algorithm design tasks may be much higher. The divide and conquer tactic, for example, only works for decompositions into two pieces. The principle of divide and conquer

is much more widely applicable, but more polymorphic than tactics can easily accommodate. The library approach used in the global search tactic for storing subspace generators is in some ways more flexible, but a designer is still limited to what someone else anticipated he would need and no library can hope to span the space of possible data structures over which search can be performed.

Tactics do too much in what is to the user one step. The global search tactic, for example, locates a suitable subspace generator, specializes it, derives filters, asks the user to select a program schema and instantiates it, all in one uninterruptible process. There are no opportunities for top-down refinement of any of the global search operations—if the tactic cannot derive a definition directly, it fails. The filter terms are more akin to program optimizations than to algorithm design, but are rather unique to global search and not achievable through the general optimizations that other KIDS operators provide. It would be desirable, however, to separate these concerns and possibly defer the filter issue, or again to be able to apply an arbitrary design tactic to the task in a top-down fashion.

Tactics, then, are not adequate design abstractions, but they contain pieces that are. An important research topic is to identify the commonalities among design tactics and abstract a set of more primitive design principles that can be easily composed in more flexible and dynamic ways to support the design of a wider class of algorithms. Some steps have already been made in this direction. An important insight made relatively early is that the abstract algorithm theories on which design tactics are based can be described in a very natural and elegant way using modern abstract algebra (50, 73).

1.2.3 Algebraic Approaches to Software Development. Algebraic specification has become the standard representation for abstract data types (6, 29), and has been promoted as a general-purpose specification language. Some examples will help illustrate the approaches taken in applying modern algebra to software development tasks. For a more complete survey, see (76).

Ehrig and Mahr develop an elaborate methodology for specifying software systems, based on equational logic and initial semantics (16, 17). A *specification* consists of a list of sorts, which represent types, a list of *operation signatures*, which represent operations on the types, and a list of equations that describe the intended behavior of the operations. An *algebra* for a specification associates with each sort a set and with each operation a function so that the signatures and equations are satisfied. A specification is taken to represent the class of all algebras that satisfy it. This definition is the *loose* semantics of a specification. In *initial* semantics, certain algebras with additional properties are distinguished as the ones intended. Ehrig and Mahr identify five characteristics an algebraic specification language should possess: *basic specifications*, which provide a means for defining specifications directly; *specification building operations*, for extending or combining existing specifications into new ones; *renaming* of specifications; *parameterization*, and *modularization*. They define a language, ACT ONE, that incorporates all of these features. A parameterized specification is actually a pair of specifications, called the *body* and the *parameter part*, related by a *specification morphism*. A specification morphism is a mapping from the sorts and operations of a *source* specification to those of a *target* specification such that equations in the source are translated by the mapping into logical consequences of the equations in the target. A parameter morphism must satisfy additional properties to insure that algebras for the body specification exist for all models of the parameter part. A parameter is *instantiated* by providing a morphism from the parameter part to some other specification (possibly the body of a parameterized specification). A basic specification for the instantiated specification can be derived by taking the *pushout* of the body and the actual parameter with respect to the parameter part. This corresponds to *amalgamation* of the algebras of the three respective specifications. A *module* is a more elaborate structure consisting of four specifications related by morphisms. The *import interface* specification defines resources needed by the module to perform its function. The *export interface* specification defines the resources provided by the module. The *parameter part* identifies parameters of the module and is related to both interface specifications by morphisms. Finally, the

body part describes how to construct the resources described in the export interface using the resources provided by the import interface; both interface specifications have morphisms to the body. Formal operations can be defined for composing modules, instantiating the parameter part, and so on. Modules support information hiding and implementation freedom (the body specification can be altered without changing the specification), and help to structure large specifications and manage their complexity. Some formal support for *refining* a specification into a lower-level but equivalent description is defined, but no automated support is provided: the ACT ONE language supports formal specification but not synthesis.

Larch is another approach to applying multi-sorted abstract algebra to software development (34). Larch is based on a two-tiered approach and represents a family of languages rather than just one. The *Larch Shared Language* (LSL) is a fairly typical algebraic specification language. Specifications are called *traits* and consist of sorts, operations and equations, as above. Traits can import other traits, but no other specification building operations are defined. Traits can be renamed to avoid name clashes with imported traits. No explicit use of morphisms is made, and parameter instantiation is not as carefully defined. There is no larger unit than the trait, such as a module. The logic of equations is extended with two special clauses: *generated by* and *partitioned by*. A generated-by clause asserts that a list of operations is a complete set of generators of a sort: that every element of the sort is represented by some composition of the listed operations. A partitioned-by clause asserts that a list of operations is a complete set of observers of a sort: that all distinct elements of the sort can be distinguished by using those operations. In a trait for sets, for example, a set of generators is the empty set and the *with* operation, and the operation $\in$ is by itself a complete set of observers. The generated-by and partitioned-by clauses assert properties of a trait that cannot be expressed in equational logic. A generated-by clause can be used to prove theorems by induction.

The intent of the shared language is to provide formal domain theories that can then be used by *interface languages* to specify actual software systems. Interface languages are extensions of standard programming languages such as C or Ada. They are used to define the components of a system, their interfaces and their behavior. Interfaces are defined according to the syntax of the base language, while behavior is described by reference to traits in the shared language. Shared language traits have a simple semantics and describe abstract components that are highly reusable. Interface specifications permit the full feature set of the base language to be used, can be compiled and executed once sufficient detail is provided, and evolve gracefully into the implemented system. No formal methods for deriving a correct implementation from a formal specification are provided, however.

OBJ3 is "a wide spectrum functional programming language ... based upon order sorted equational logic" (30). Order sorted means that sorts can be declared to be subsorts (or subtypes) of other sorts. This provides a formal mechanism for exception handling and for handling multiple inheritance in object-oriented systems. OBJ3 defines two kinds of modules: *objects* and *theories*. An object is used to encapsulate executable code (written in LISP) by defining the sorts and operations provided and describing their properties in equational logic. A theory is an object with no associated code, corresponding to a specification in ACT ONE or a trait in LSL. Both theories and objects can be parameterized. A *view* is essentially a morphism, showing how one module possesses all the properties of another. The only object or theory building operation is *import*. Module renaming is supported. The equations in a theory can be treated as rewrite rules and thus executed after a fashion; the code associated with an object can be executed directly. OBJ3 is described as a programming language, but an executable specification is not an industrial-strength program and there seems to be little support for refining abstract specifications to traditional algorithms or data structures.

1.2.4 SPECWARE. SPECWARE is a follow-on system to KIDS, DTRE and REACTO (22) dedicated to the proposition that all software tasks from specification and design to implementation and maintenance can be performed entirely by algebraic means (43). This proposition is far from proved, but lessons learned from SPECWARE promise to enhance our understanding of software and our ability to apply formal methods to its construction and evolution.

The SPECWARE view of algebraic specification is based on and motivated by the more fundamental concepts of category theory. Category theory was developed as a means to describe and characterize certain recurring mathematical structures within a unified framework (53, 77). A category consists of a class of *objects*, a function associating with each pair of objects a set of *arrows* between them, and a small number of axioms that the objects and arrows have to satisfy. The class of algebraic specifications and the morphisms between them are, respectively, the objects and arrows of a category. The algebras of a given specification and the homomorphisms between them also form a category. Several other categories arise in the context of software development. Category theory provides a uniform theoretical framework for describing and relating these structures.

SPECWARE makes heavy use of the category theoretic notion of a *diagram*. A diagram is a directed multi-graph in which the nodes and arcs are labeled with objects and arrows, respectively, from some category. Diagrams made from specifications and specification morphisms provide a means for describing a system as an assemblage of components rather than by a single, monolithic specification. The *colimit* of a diagram is an object that is in a certain sense equivalent to the diagram. Since a colimit is an object it can be used in diagrams, allowing hierarchical structures to be built. This approach is much more general than import, which becomes a special case.

SPECWARE provides a specification language called Slang (80). Slang provides a very rich, higher-order logic for writing basic specifications; it is not limited to equational logic as the systems above are. Specification building operations include import, translation (renaming via an isomorphism) and colimit. SPECWARE also provides a graphical editor for manipulating diagrams visually.

Parameterized specifications are not directly supported (yet), but can largely be simulated. The pushout operation mentioned above, for example, is just the colimit of a particular diagram shape.

SPECWARE supports not only formal specification of systems but also a formal model of refinement. During specification the user is free to describe the system at any level of abstraction, but in general more abstract descriptions are preferred, to help keep the specification relatively small and to constrain the eventual implementation as little as possible. Refinement incrementally transforms a specification at one level of abstraction into a lower level. Once a specification has been sufficiently refined, it can be used to generate code. Specification and refinement may be intermixed; the distinction is primarily logical, not temporal.

The basic unit of refinement in SPECWARE is the *interpretation*. An interpretation is a generalization of a morphism. Instead of mapping symbols of a source specification to symbols of a target, symbols are mapped to arbitrary expressions built by composing target symbols. These expressions are identified and named in an explicit *mediator* specification that imports the target specification and has a morphism from the source specification. The import morphism from the target specification to the mediator is called a *definitional extension* because it adds only definitions (named expressions) that in some sense were already there, rather than anything new. Interpretations can be used to show how one specification implements another. For example, a specification for finite sets can be refined into a specification for bit vectors, the latter being much closer to a machine-level concept.

Interpretations can also be treated as objects and a suitable notion of *interpretation morphism* defined. Diagrams and colimits are defined for interpretations and interpretation morphisms in much the same way as they are for specifications and specification morphisms. SPECWARE implements a technique called *diagram refinement* that allows a specification to be refined by exploiting its structure: just as a specification can be constructed as the colimit of a diagram of specifications and morphisms, each of these can be refined, separately but compatibly, with interpretations and

interpretation morphisms, respectively, and the colimit of this diagram computed, forming a refinement of the original specification. Structured refinement helps to control the complexity of the design process and enhances the ability of the SPECWARE approach to scale up to large systems.

SPECWARE generates code from specifications by formalizing a *logic* for the target programming language as a category of specifications, morphisms and deduction rules. This logic need not look anything like Slang. Currently SPECWARE contains logics for a functional subset of LISP and a functional subset of C++ (functional subsets are easier to work with because Slang is a functional language). *Interlogic morphisms* provide a means for translating from Slang to the target language. These morphisms describe not only how symbols are translated but also how the logic of Slang itself (sorts, operation signatures, logical operations such as quantifiers, etc.) are translated. For example, LISP is an untyped language, so each sort in Slang is translated to a LISP predicate, and where axioms in Slang use logical variables of a sort, the translated LISP axioms use untyped variables and explicit tests. A pretty printer is used to take target language specifications and convert them to their conventional surface syntax, which can then be compiled and executed.

Since the original specification and its refinement are all explicit, first-class objects, the entire design history of the system is preserved. Changes can be made and propagated through this structure in order to perform maintenance. Thus SPECWARE seems capable of supporting the entire software lifecycle.

SPECWARE provides formal mechanisms for the major software development tasks of specification, design and implementation. What it lacks at this stage of its development is automated support for the creative processes involved in specification acquisition and software design. One approach to the former is DeLoach's work on integrating algebraic specification with Rumbaugh's Object Modeling Technique (13). DeLoach defines a two-way translation between object modeling concepts (classes, associations, events, states, aggregation, inheritance, etc.) and an extension of Slang called O-Slang. This translation implicitly defines a formal semantic for object modeling,

while retaining all of its desirable features in terms of intuition and a popular diagramming notation: the user works exclusively with an object-oriented graphical interface and can be completely oblivious to the formal manipulations occurring behind the scenes. O-Slang is easily translated into Slang and the resulting system specification can then be refined using SPECWARE.

Design and implementation are refinement processes, and refinement is an arbitrarily difficult problem. Techniques need to be developed for representing and manipulating design knowledge of all kinds, including software architectures, algorithms, data structures and so on. As a beginning, one would want to take what DTRE and KIDS do and implement it in SPECWARE. My research is a contribution toward implementing the algorithm design tactics of KIDS in a SPECWARE-like system. It builds on some preliminary work that has already been done and which is described next.

1.2.5 Algebraic Algorithm Design. The examples above clearly illustrate that casting algorithm design into an algebraic framework brings a lot of theoretical and practical knowledge to bear on the problem. In the algebraic approach to algorithm design, the components of an algorithm are defined as abstract sorts and operations, and the constraints among these components are written as axioms (73). A specification for global search, for example, is shown in Figure 2, which is based on (69). $\hat{R}$ is the sort for subspace descriptors, and the predicate $\hat{I}$ decides which descriptors are legal. *Satisfies* relates solutions to subspaces, defining how the former satisfy the constraints implicit in the latter. The other sorts and operations were described earlier. Axiom *GS0* states that the initial subspace descriptor generated for legal inputs is legal. Axiom *GS1* states that when a legal descriptor is split, the new descriptor is also legal. Axiom *GS2* states that all legal solutions are contained in or satisfy the initial descriptor. Axiom *GS3* states that all the solutions satisfying a particular descriptor can be extracted after a finite number of splits.

The primary task of a design tactic is to construct an interpretation from an algorithm theory specification to a problem specification. Smith has identified several general methods for

```

spec GlobalSearch is
  sorts  $D, R, \hat{R}$ 
  op  $I : D \rightarrow \text{Boolean}$ 
  op  $O : D, R \rightarrow \text{Boolean}$ 
  op  $\hat{I} : D, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\hat{r}_0 : D \rightarrow \hat{R}$ 
  op  $Satisfies : R, \hat{R} \rightarrow \text{Boolean}$ 
  op  $Split : D, \hat{R}, \hat{R} \rightarrow \text{Boolean}$ 
  op  $Extract : R, \hat{R} \rightarrow \text{Boolean}$ 

  axiom GS0 is  $\forall(x : D) (I(x) \Rightarrow \hat{I}(x, \hat{r}_0(x)))$ 
  axiom GS1 is  $\forall(x : D, \hat{r}, \hat{s} : \hat{R}) (I(x) \wedge \hat{I}(x, \hat{r}) \wedge Split(x, \hat{r}, \hat{s}) \Rightarrow \hat{I}(x, \hat{s}))$ 
  axiom GS2 is  $\forall(x : D, z : R) (I(x) \wedge O(x, z) \Rightarrow Satisfies(z, \hat{r}_0(x)))$ 
  axiom GS3 is  $\forall(x : D, z : R, \hat{r} : \hat{R}) (I(x) \wedge \hat{I}(x, \hat{r})$ 
     $\Rightarrow (Satisfies(z, \hat{r}) \Leftrightarrow \exists(\hat{s}) (Split^*(x, \hat{r}, \hat{s}) \wedge Extract(z, \hat{s})))$ 
    where  $Split^*$  is the reflexive and transitive closure of  $Split$ 
end-spec

```

Figure 2. Specification of the Global Search Algorithm Theory

constructing interpretations automatically (72). One of these is *unskolemization*, a theorem proving technique that can derive a definition for an operation from axioms that relate its intended behavior to other operations whose definitions are known. A logical formula is *skolemized* by replacing existentially quantified variables with constants or calls to functions. Unskolemization is the inverse process of replacing calls to unknown functions with existentially quantified variables. If a formula contains only one function symbol that is undefined, unskolemization yields a theorem that can be proved and the existential variable acts as a *witness* for the theorem that can be used to provide a definition for the function that the variable replaced. The tactic for divide and conquer derives a definition for *Compose* from one given for *Decompose*, or vice versa, by unskolemizing axiom 1, above, that relates them.

Another interpretation construction technique uses what are called *connections between specifications*. A connection is a pair of interpretations with a common source specification together with a set of *conversion functions* between corresponding sorts in the two target specifications; these conversion functions must satisfy a set of *connect conditions* that relate corresponding operations in the two targets. A theorem by Smith states that if only one of the interpretations is known to

be valid (that is, the mapping from the source to the mediator is a valid specification morphism), and if conversion functions are defined such that all the connection conditions hold, then the other interpretation must also be valid (72).

Connections can be used to construct an interpretation as follows. The global search design tactic uses this technique, so it will be used as an example. First a library of interpretations from some source specification is developed. The library of subspace generators serves exactly this purpose. To build a new interpretation to some problem specification, we extend the known mapping for D , R , I and O by adding to the mediator specification the additional sorts and operations of the algorithm theory, but without definitions (thus the mediator is no longer a definitional extension of the target specification, and the mapping from the algorithm theory is not truly a morphism). We then add conversion functions and connection conditions. The connection conditions can be unskolemized to provide definitions for the conversion functions. These conversions are then used to define operations for the new interpretation in terms of the operations of the library interpretation. By Smith's theorem above, the morphism to the mediator is a valid specification morphism, and since all the symbols added to the mediator now have definitions, it is also a definitional extension of the target.

Formula 2 shown earlier for the global search tactic of KIDS turns out to combine pieces of several connection conditions. The expression $R_A = R_B$ means that the conversion for R is the identity function, and the rest of it is essentially the connection condition for O . The only undefined operation is the conversion function from D_B to D_A , shown already unskolemized as the variable y . The function for $\hat{R}$ is also an identity function, so solving the condition shown for y gives definitions for all the conversion functions, and this provides enough information to complete the interpretation from global search based on the selected subspace generator. Filter terms are derived separately, by another theorem-proving technique. Chapter V contains a fuller explanation

```

spec Problem is
  sorts D, R
  op I : D → Boolean
  op O : D, R → Boolean
end-spec

```

Figure 3. Specification of Basic Problem Theory

of the connection mechanism and shows in detail how the construction would be performed in SPECWARE.

Smith has also devised an incremental approach to refinement that he calls the classification approach to design (71). He shows how a refinement hierarchy of *design theories* can be used to refine a specification incrementally, which can simplify the construction of interpretations. A design theory may directly relate to solution synthesis, as we've seen for algorithm theories, or may instead serve only to classify a problem by showing how it is an instance of a known problem class. In either case, an interpretation from a design theory to a problem specification identifies structure in the problem that can be exploited to solve it. The key idea is that design theories are often themselves refinements of other design theories, and that organizing them into a hierarchy based on this structure provides an efficient way to explore approaches to a problem formally and systematically. At the top of the hierarchy is a specification, *Problem*, that contains the essential elements of a problem specification with no additional structure specified. This specification is shown in Figure 3. Immediate refinements of *Problem* include very general problem classes such as constraint satisfaction problems and general algorithm theories such as global search. Constraint satisfaction encompasses many more specific problems such as combinatorial satisfaction, integer programming and real-valued programming, and each of these in turn has further refinements. Global search can be refined by adding structure needed to do constraint propagation (75).

A problem is solved most efficiently by exploiting as much of its structure as possible, that is, by classifying it as deeply in the refinement hierarchy as we can. Classification begins at the root of the hierarchy, where an interpretation from *Problem* is given by the problem specification

itself. All refinements of *Problem* are candidates for the next level of classification. These are all very general problem classes or theories that identify common problem structures. Once an interpretation from one or more of these is constructed, refinements at the next level down become candidates. Each classification step adds one "rung" to a ladder of refinements; each increment adds a little more structure to the one before, keeping construction of each interpretation relatively easy. Figure 4 shows an example of the process that starts by classifying a problem *MyProblem* as a constraint satisfaction problem and ends with an unfinished attempt to classify it as network flow. The double arrows represent interpretations, with the mediators not shown. When no further refinements can be found, the problem has been fully classified (and possibly in more than one way, if more than one classification attempt at a given level succeeds). At this point a program schema can be applied to generate an algorithm for the problem. For problem classes such as linear programs or network flow, for which efficient software already exists, the derived algorithm may do nothing more than set up a call to a commercial solver. This represents one method for using formal derivational techniques to leverage existing software investments.

These examples suggest that unskelemization and reasoning about connections, combined with simpler techniques such as composition of interpretations, could form the basis for refinement operators that can be made available directly to the designer rather than hidden within tactics. Doing so reduces the need for tactics substantially by providing their primary function via general mechanisms rather than custom code. My research continues the effort by defining explicitly the representations and manipulations needed to apply algorithmic knowledge to a design problem in this fashion.

1.2.6 Local Search. The literature on local search, primarily from the operations research community, is vast but largely informal. Much of it takes the form of descriptions of particular algorithms and the results of computational experiments. A good introduction to the technique is

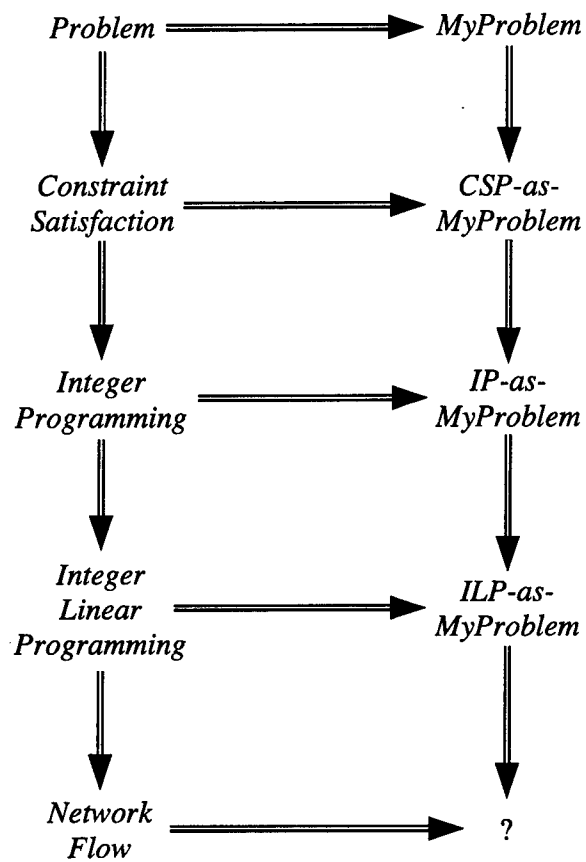


Figure 4. Ladder Diagram for Classification Approach to Design

in (59). Some specific algorithms and general approaches will be presented in Chapter III. A few theoretical results, mostly related to computational complexity, will be described in Chapter VI.

The only work on local search related to software synthesis is by Mike Lowry. Lowry uses local search to illustrate his formal notion of problem reduction (50) and the process by which algorithm theories and design tactics are created, which he calls design analysis (51). These two renditions of his work on local search are not completely consistent with each other or with the tactic as he implemented it in KIDS, and developments since then have shed new light on how to interpret and carry out such work, but the majority of the work that Lowry did remains sound.

In (51) Lowry gives a specification for what he calls abstract local search theory, shown in Figure 5 as it would appear more or less in Slang (more detail on the notation and terminology of Slang is in Chapter II). It contains a specification for an optimization problem, which consists of a feasibility problem described by D , R , I and O and a cost function over solutions. Lowry calls O the *feasibility condition* since it characterizes the space of feasible solutions over which we are optimizing. The specification also includes a *neighborhood relation*, N . There are three axioms concerning N , with all but the first being optional. The first one requires that each solution be a neighbor of itself. Since this would permit a search to “move” from a solution to itself, reflexivity seems rather to be something to avoid. The second axiom defines *exactness* of a neighborhood, which is that if all the neighbors of a solution are no better in cost, then that solution is a global optimum. The third defines *reachability*, which is that from any solution one can reach any other by a sequence of neighbors. This specification captures what Lowry considers to be the essence of local search, abstracting away details about specific approaches.

Because of its abstractness and generality, abstract local search theory is too weak to support the construction of a concrete algorithm. In particular, it provides little help in the critical task of constructing a neighborhood relation suited to a particular problem. Lowry therefore specializes it by providing a theory (in the general sense of the word) of a specific class of neighborhood

```

spec AbstractLocalSearchTheory is
  import translate TotalOrder by  $\{E \rightarrow \mathcal{R}\}$ 

  sorts  $D, R$ 

  op  $I : D \rightarrow \text{Boolean}$ 
  op  $O : D, R \rightarrow \text{Boolean}$ 
  op  $N : D, R, R \rightarrow \text{Boolean}$ 
  op  $Cost : D, R \rightarrow \mathcal{R}$ 

  op  $Optimal : D, R \rightarrow \text{Boolean}$ 
  definition of  $Optimal$  is
    axiom  $Optimal(x, y) \Leftrightarrow \forall(y') (O(x, y') \Rightarrow Cost(x, y) \leq Cost(x, y'))$ 
  end-definition

  axiom ReflexiveNeighborhood is
     $\forall(x, y : D) (I(x) \wedge O(x, y) \Rightarrow N(x, y, y))$ 

  axiom ExactNeighborhood is
     $\forall(x, y : D) (I(x) \wedge O(x, y) \wedge \forall(y') (O(x, y') \wedge N(x, y, y') \Rightarrow Cost(x, y) \leq Cost(x, y'))$ 
       $\Rightarrow Optimal(x, y))$ 

  axiom ReachableNeighborhood is
     $\forall(x, y, y' : D) (I(x) \wedge O(x, y) \wedge O(x, y') \Rightarrow N^*(x, y, y'))$ 
    where  $N^*$  is the reflexive and transitive closure of  $N$ 
end-spec

```

Figure 5. Abstract Local Search Theory

relations that are commonly used. This is a descriptive theory only; it does not provide a means for constructing neighborhoods automatically. Lowry uses it to guide the population by hand of a small library of neighborhoods over Refine data types.

Lowry then describes a design tactic based on abstract local search theory that automates the construction of local search algorithms tailored to particular problems. The tactic essentially uses the connection mechanism to specialize one of the library neighborhoods to a problem specification. It next does some derivations designed to enhance the efficiency of the eventual algorithm. It then instantiates the only program scheme that Lowry provides, which implements hill-climbing: starting from an initial solution, the search progresses by choosing at each step a move to a solution of better cost, until no such moves exist. A function for computing an initial solution is specified but not defined; this function can be refined using any of the KIDS design tactics. Other schemas could,

Lowry claims, be used to create other kinds of local search algorithms from the same underlying components.

Lowry demonstrates the usefulness of his theory and tactic by applying it to linear programming. The algorithm generated is a variant of the simplex, one of the best-known and well-studied algorithms around.

Lowry's theory is neither complete nor correct. His axiomatization of neighborhood properties is inadequate; it includes some that are inappropriate and misses others that the tactic and program scheme in fact enforce, but too rigidly. The tactic as a whole seems specialized to the hill-climbing program scheme and even to the simplex derivation. The theory of basic neighborhoods is interesting but ultimately fails to assist the process of constructing good neighborhoods. Lowry's claim that the theory and tactic are adequate for most or all local search algorithms is unsubstantiated. These criticisms and others are discussed more in Chapter VI, where a detailed analysis of the tactic is undertaken, and in Chapter VII, which demonstrates some specialized local search techniques that depend on identification of additional problem structure. In spite of the flaws, however, Lowry's algorithm theory and design tactic for local search provide a good example of the KIDS approach to algorithm design, and served as a starting point in my own formalization of local search.

1.3 Overview of Document

This chapter has introduced the goals, objectives and contributions of my research and outlined related work in knowledge-based software engineering in general and transformational systems, algebraic approaches and local search in particular. The remainder of this dissertation is organized as follows:

- Chapter II explains the terminology and notation of SPECWARE and the language Slang, which will be used throughout the rest of the document.

- Chapter III presents an informal framework for describing and classifying local search algorithms.
- Chapter IV describes a general approach to algorithm design by algebraic means, formalizing the approach taken in KIDS and adding structures that insure the correctness of the result.
- Chapter V describes techniques for performing some of the constructions described in Chapter IV. The main result is an algebraic theory for constructing *connections between specifications*.
- Chapter VI defines a basic theory for local search, focusing on neighborhood structures and their properties. It describes a tactic for adapting neighborhood structures from a library to particular problems. It then formalizes two local search techniques, hill climbing and simulated annealing, that the basic theory is sufficient to describe.
- Chapter VII specializes the basic theory defined in Chapter VI by adding additional sorts and operations needed to support two advanced local search techniques, tabu search and the Kernighan-Lin heuristic.
- Chapter VIII provides conclusions evaluating the work done and summarizing its contributions. It also gives some suggestions for future work in this area.
- Appendix A is a library of neighborhood structures for the theory developed in Chapter VI. It also includes discussion of some topics introduced in Chapter VII.
- Appendix B contains a theorem and its proof concerning a property of the classification diagrams introduced in Chapter IV. This is an interesting technical result that is not central to the theory of algebraic algorithm design.
- Appendix C critiques Lowry's theory of *basic neighborhoods* (51), an attempt to characterize formally a class of neighborhoods.

II. Notation and Terminology

The algebraic language used in this dissertation is Slang, the language of the SPECWARE system. The definitive reference on Slang is the SPECWARE Language Manual (80). This chapter presents much of the syntax of Slang along with informal, intuitive explanations of its many features and examples that will contribute to later chapters. The syntax of Slang is LISP-like and somewhat hard to read, however, so a richer and more natural syntax will be adopted, in particular when writing logical formulas (this variant might be termed euSlang). Most of the examples are taken from or based on examples in the Language Manual or specs that come with distributions of SPECWARE.

The features of Slang are described in two main sections based on their primary intended use: specification and refinement. Features of Slang unique to code generation are not described at all. Slang has a strong category theory flavor, and some category theory is presented along with the language. Additional references for category theory include (32, 53, 77).

2.1 Specification Constructs

The specification features of Slang allow a user to describe a system at some chosen level of abstraction. These features support a compositional style, building up complex descriptions by composing simpler ones in formal ways.

2.1.1 Specs. The fundamental object in Slang is the *specification*, or simply *spec* (the word *specification* is hereby reserved for more generic purposes). A spec consists of a set of *sorts*, which represent abstract types, a set of *operations*, and a set of *axioms*, which are logical statements written in higher-order logic that describe or constrain the behavior of the operations. Figure 6 is an example of a *basic spec*, which is a spec that explicitly presents its sorts, operations and axioms. *TotalOrder* introduces one sort, *E*, which represents some unspecified type or set, and one operation, the binary relation $\leq$, which compares two elements of sort *E* and returns a boolean

```

spec TotalOrder is
  sort E

  op  $\_ \leq \_$  : E, E  $\rightarrow$  Boolean

  axiom reflexivity is  $\forall (x : E) (x \leq x)$ 
  axiom anti-symmetry is  $\forall (x : E, y : E) (x \leq y \wedge y \leq x \Rightarrow x = y)$ 
  axiom transitivity is  $\forall (x : E, y : E, z : E) (x \leq y \wedge y \leq z \Rightarrow x \leq z)$ 
  axiom totality is  $\forall (x : E, y : E) (x \leq y \vee y \leq x)$ 
end-spec

```

Figure 6. A Basic Spec for a Generic Total Order

value. The axioms assert properties of $\leq$. A sort is identified by giving its name. An operation is identified by giving its name and its *signature*, which is the sort to which it belongs, typically a *function sort* that specifies a *domain* and a *codomain* sort. The underscores ‘_’ around the name are used to suggest that infix notation is preferred when this operation is used. The sort *Boolean* and the standard logical operations ($\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$, $\neg$, the constants *true* and *false*, and equality) are built into Slang and present in all specs. The empty spec,

```

spec Boolean is end-spec

```

which introduces no sorts, operations or axioms, contains only what is built in.

Figure 7 is a basic spec for the natural numbers that introduces a number of additional features. First, the signature of *zero* is *Nat*, not a function sort. Such an operation is called a *constant*. Second, the axiom describing *nonzero?* is presented as a *definition*. When one or more axioms is enclosed in a definition clause, this asserts that the axioms completely describe all of the behavior of the operation: that is, that a provably unique relationship has been imposed between inputs and outputs. Third, the meaning of the sort *Pos* is constrained by use of a *sort axiom*. A sort axiom equates a *primitive* sort with a *constructed* sort. A primitive sort is just a name. A constructed sort is a term consisting of any of Slang’s built-in *sort constructors*. It is not legal for a sort axiom to equate two constructed sorts or for multiple axioms to equate a primitive sort with more than one constructed sort, though future versions of SPECWARE may support these

```

spec Nat is
  sorts Nat, Pos

  op zero : Nat
  op nonzero? : Nat → Boolean
  definition of nonzero? is
    axiom  $\forall (k : \text{Nat}) (\text{nonzero?}(k) \Leftrightarrow \neg(k = \text{zero}))$ 
  end-definition

  sort-axiom Pos = Nat | nonzero?

  op nat-of-pos : Pos → Nat
  definition of nat-of-pos is
    axiom nat-of-pos = (relax nonzero?)
  end-definition

  op succ : Nat → Pos
  axiom  $\forall (k : \text{Nat}) \text{nonzero?}(\text{nat-of-pos}(\text{succ}(k)))$ 
  axiom  $\forall (j : \text{Nat}, k : \text{Nat}) (\text{succ}(j) = \text{succ}(k) \Rightarrow j = k)$ 

  constructors {zero, nat-of-pos} construct Nat
  constructors {succ} construct Pos

  op _ ≤ _ : Nat, Nat → Boolean
  definition of ≤ is
    axiom  $\forall (k : \text{Nat}) (k \leq \text{zero} \Leftrightarrow k = \text{zero})$ 
    axiom  $\forall (j : \text{Nat}, k : \text{Nat})$ 
       $(j \leq \text{nat-of-pos}(\text{succ}(k)) \Leftrightarrow j \leq k \vee j = \text{nat-of-pos}(\text{succ}(k)))$ 
  end-definition

  op one : Pos
  definition of one is
    axiom one = succ(zero)
  end-definition
end-spec

```

Figure 7. A Basic Spec for the Natural Numbers

capabilities. A sort axiom is analogous to a definition: in each case, the sort or operation has been completely described.

Slang provides five constructors for building up new sorts from existing ones. Sort constructors implicitly define certain built-in operations, and axioms that define them. The constructors and their associated operations are as follows:

- A *product sort* is denoted by a list of sorts separated by commas. Elements of a product sort are *tuples* consisting of a list of values, each belonging to the corresponding sort. Product sorts are analogous to cross products of sets. A product sort implicitly defines projections

$$\text{op}(\text{project } i) : S_1, \dots, S_n \rightarrow S_i$$

that map tuples to individual components.

- A *coproduct sort* is denoted by a list of sorts separated by $+$. Values of a coproduct sort consist of all the values of any of the component sorts. Coproducts are analogous to the disjoint union of sets. A coproduct sort implicitly defines embeddings

$$\text{op}(\text{embed } i) : S_i \rightarrow S_1 + \dots + S_n$$

that map values of each component sort into the coproduct.

- A *function sort* is denoted by an expression of the form $D \rightarrow C$. As mentioned above, most operations belong to function sorts. An unnamed operation can be denoted $\lambda(d : D) \text{exp}$ where d is a variable of sort D and exp computes a value of sort C .
- A *subsort* is denoted by an expression of the form $S \mid P$, where $P : S \rightarrow \text{Boolean}$ is an operation in the spec. The values of $S \mid P$ are those values of S for which P is *true*. A subsort is analogous to a subset, where P is the characteristic function defining it. A subsort implicitly defines a relaxation

$$\text{op}(\text{relax } P) : S \mid P \rightarrow S$$

that maps values of the subsort to values of the base sort.

- A *quotient sort* is denoted by an expression of the form S/Q , where $Q : S, S \rightarrow \text{Boolean}$ is an equivalence relation on S (that is, it must be provably reflexive, symmetric and transitive). The values of S/Q are equivalence classes induced by Q on S . A quotient sort implicitly defines an abstraction

op (quotient Q) : $S \rightarrow S/Q$

that maps values of S to their equivalence classes in S/Q .

Returning to the spec *Nat*, the *constructor* clause asserts that all the values of a specified sort are in the range of one or more of the operations listed. A constructor clause implicitly defines an induction axiom: if a property holds for all values in the ranges of the constructors, then it holds for all values of the sort. It does not state that all such values are unique, however: other axioms are required to assert or deny this. For *succ*, the second axiom states that all values are unique; that is, that *succ* is injective. The first axiom states that no value of sort *Pos* corresponds to the natural number *zero*. These two axioms plus the induction axioms implied by the constructor clauses are equivalent to Peano's axioms for the natural numbers.

Figure 8 shows a basic spec for finite maps that illustrates another use of subsorts. A finite map associates for some set of elements of sort *Dom* elements of sort *Cod*. The operation *map-apply*, which returns the values previously assigned, is only defined at a particular *Dom* element x if x was previously assigned a value by *map-shadow*. In Slang, all operations are assumed to be total, that is, defined over their entire domain sort. Therefore a subsort is used to restrict the domain of *map-apply* to only those pairs of maps and *Dom* elements to which values have been assigned. The subsort is anonymous, appearing only in the signature of *map-apply*. The definition of *map-apply* illustrates some of the inconvenience that can arise when using subsorts. The **relax** operation that comes automatically with a subsort maps an element of the subsort to the base sort. There is no operation for converting elements of the base sort that satisfy the subsort predicate into the subsort itself; nor can there be, for such an operation would not be defined for elements that did not satisfy the subsort predicate: its domain would be restricted to the subsort itself! The technique that is most often used is to declare a variable of the subsort and assert that its relaxation is equal to some value of the base sort that satisfies the subsort predicate, and then let this equality imply the desired result. We will see many examples of this technique, primarily when *FiniteMap* is used.

```

spec FiniteMap is
  sorts Dom, Cod, Map

  op empty-map : Map
  op map-shadow : Map, Dom, Cod → Map

  constructors {empty-map, map-shadow} construct Map

  axiom empty-map-is-distinct is
     $\forall (x : \text{Dom}, y : \text{Cod}, M : \text{Map}) \neg (\text{empty-map} = \text{map-shadow}(M, x, y))$ 
  axiom duplicate-shadow-overrides is
     $\forall (x : \text{Dom}, y1 : \text{Cod}, y2 : \text{Cod}, M : \text{Map})$ 
       $(\text{map-shadow}(\text{map-shadow}(M, x, y1), x, y2) = \text{map-shadow}(M, x, y2))$ 
  axiom independent-shadows-commute is
     $\forall (x1 : \text{Dom}, x2 : \text{Dom}, y1 : \text{Cod}, y2 : \text{Cod}, M : \text{Map})$ 
       $(\neg (x1 = x2) \Rightarrow \text{map-shadow}(\text{map-shadow}(M, x1, y1), x2, y2)$ 
         $= \text{map-shadow}(\text{map-shadow}(M, x2, y2), x1, y1))$ 

  op defined-at? : Map, Dom → Boolean
  definition of defined-at? is
    axiom  $\forall (x : \text{Dom}) \neg \text{defined-at?}(\text{empty-map}, x)$ 
    axiom  $\forall (x1 : \text{Dom}, x2 : \text{Dom}, y : \text{Cod}, M : \text{Map})$ 
       $(\text{defined-at?}(\text{map-shadow}(M, x1, y), x2) \Leftrightarrow x1 = x2 \vee \text{defined-at?}(M, x2))$ 
  end-definition

  op map-apply : (Map, Dom) | defined-at? → Cod
  definition of map-apply is
    axiom  $\forall (x : \text{Dom}, y : \text{Cod}, M : \text{Map}, M\text{-at-}x : (\text{Map}, \text{Dom}) \mid \text{defined-at?})$ 
       $(\langle \text{map-shadow}(M, x, y), x \rangle = (\text{relax } \text{defined-at?})(M\text{-at-}x)$ 
         $\Rightarrow \text{map-apply}(M\text{-at-}x) = y)$ 
  end-definition
end-spec

```

Figure 8. Basic Spec for Finite Maps

```

spec MinMax is
  import TotalOrder

  op min : E, E → E
  definition of min is
    axiom  $\forall(x : E, y : E) (x \leq y \Rightarrow \text{min}(x, y) = x)$ 
    axiom  $\forall(x : E, y : E) (y \leq x \Rightarrow \text{min}(x, y) = y)$ 
  end-definition

  op max : E, E → E
  definition of max is
    axiom  $\forall(x : E, y : E) (x \leq y \Rightarrow \text{max}(x, y) = y)$ 
    axiom  $\forall(x : E, y : E) (y \leq x \Rightarrow \text{max}(x, y) = x)$ 
  end-definition
end-spec

```

Figure 9. Import Example

For large specs, explicitly listing all of the sorts, operations and axioms is inconvenient: difficult to write, difficult to read and understand. Slang provides a number of mechanisms for controlling complexity by structuring specs. The first of these is import. Figure 9 shows a spec that imports *TotalOrder* and then extends it by adding two operations and their definitions. In general, extensions can introduce any kind of spec element: sorts, sort axioms, operations, axioms, constructors, definitions, theorems, etc. The spec *MinMax* is equivalent to a basic spec that contains all the elements of *TotalOrder* followed by the additional elements in *MinMax*.

A spec is a finite presentation of an infinite object, called a *theory*. The theory *generated by* a spec consists of the built-in sorts, operations and axioms, the explicitly named sorts, operations and axioms, any implicit operations or axioms as described above, all theorems that follow logically from the axioms, and all possible sorts and operations that can be defined (using sort axioms and definitions) in terms of other sorts and operations in the theory.

2.1.2 Morphisms. The second most fundamental object in Slang is the *morphism*. A morphism is a function that maps the sorts and operations of one spec, called the *source* or *domain*, to those of another spec, called the *target* or *codomain*, in such a way that the axioms of the source spec are translated by the morphism into theorems in the target spec. A morphism shows how the

structure described by one spec is also present in another. The target spec may in fact have more structure, but it at least contains the structure of the source. For example,

morphism *TO-to-Nat* : *TotalOrder* $\rightarrow$ *Nat* is $\{E \rightarrow Nat, \leq \rightarrow \leq\}$

shows that the $\leq$ relation defined for natural numbers is a total order. This morphism imposes a proof obligation that the axioms of total order follow logically from the axioms and definitions in *Nat*. A morphism is represented graphically by drawing an arrow between two specs.

Like a spec, a morphism is a finite presentation of an infinite object, one that maps all the elements of one theory into those of another. Only primitive sorts and operations (those identified by a name) are explicitly assigned mappings. Built-in sorts and operations are always mapped automatically to themselves, and terms are mapped in the obvious way by recursively translating each element from which the term is constructed. As a shortcut, if a source element is to be mapped to a target element of the same name, the mapping may be omitted in the presentation of the morphism and SPECWARE will fill it in automatically. In *TO-to-Nat*, for example, the mapping for $\leq$ could be omitted.

Whenever one spec imports another, a morphism is implicitly created. This morphism can be referred to by a keyword. For example, to give the import morphism for *MinMax* a name, one would write

morphism *TO-to-MinMax* : *TotalOrder* $\rightarrow$ *MinMax* is import-morphism

Morphisms form the basis of a second spec building operation, translation. The *translate* operation makes a copy of a spec and optionally renames its elements. To create a copy of *Nat* for use as a counter, for example, one would write

spec *Counter* is translate *Nat* by $\{Nat \rightarrow Count\}$

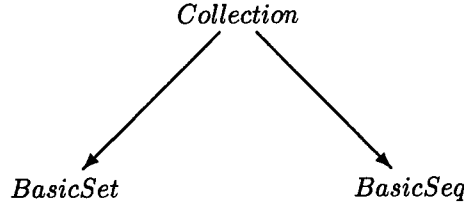
Like import, *translate* implicitly creates a morphism that is referred to by a keyword:

morphism *Nat-to-Counter* : $Nat \rightarrow Counter$ is translation-morphism

Translation and import can be combined in one step. Figure 10 shows a spec for a collection. This spec captures the common features of such collections as sets, bags and sequences. Collections are constructed from the empty collection and the insert operation. The spec does not specify whether the order in which elements are inserted is preserved, as it is in sequences but not sets or bags, or whether multiplicity of inserted elements is preserved, as in sequences and bags but not sets. In fact, it even permits such degenerate cases as all collections being equal to the empty collection.

To specify sets, *Collection* is imported and additional elements are added to constrain the operations to behave as desired. To adopt names that are conventional for sets, *Collection* is translated before it is imported. The import clause imports an anonymous spec generated by the translate clause. The *in* operation permits the contents of a set to be viewed. Its axioms imply that the empty set is distinct from non-empty sets and that elements inserted into collections remain there. The exchange axiom states that the order of insertions is irrelevant, and the condensation axiom states that multiplicity is irrelevant. From these axioms the standard definition of set equality follows, that two sets are equal if and only if they contain the same elements.

To specify sequences, *BasicSeq* imports *Collection* and adds a different set of extensions. No operations are added, just two axioms. The first asserts that the empty sequence is not equal to the result of an insertion. Note that this axiom was a theorem for sets. The second axiom asserts that the order (and hence multiplicity, in a sense) of insertions is significant. The translation renames the insertion operation to *prepend*, which is thought of as the operation that adds an element to the front of the list, but this is just convention: nothing in the spec so far distinguishes any part of a sequence from any other, front, back or middle.



```

spec Collection is
  sorts E, Col

  op empty-col : Col
  op insert : E, Col → Col

  constructors {empty-col, insert} construct Col

  sort NE-Col
  sort-axiom NE-Col = Col | nonempty-col?

  op nonempty-col? : Col → Boolean
  definition of nonempty-col? is
    axiom  $\forall(C : Col) (nonempty-col?(C) \Leftrightarrow \neg(C = empty-col))$ 
  end-definition
end-spec

spec BasicSet is
  import translate Collection by {Col → Set, NE-Col → NE-Set, empty-col → empty-set,
    nonempty-col? → nonempty-set?}

  op in : E, Set → Boolean
  axiom empty is  $\forall(e : E) \neg in(e, empty-set)$ 
  axiom in is  $\forall(d : E, e : E, S : Set) (in(e, insert(d, S)) \Leftrightarrow e = d \vee in(e, S))$ 

  theorem retention is  $\forall(e : E, S : Set) (in(e, insert(e, S)))$ 
  theorem conservation is  $\forall(e : E, S : Set) \neg(empty-set = insert(e, S))$ 

  axiom exchange is
     $\forall(d : E, e : E, S : Set) (insert(e, insert(d, S)) = insert(d, insert(e, S)))$ 
  axiom condensation is  $\forall(e : E, S : Set) (insert(e, insert(e, S)) = insert(e, S))$ 
  theorem equality is  $\forall(S : Set, T : Set) (S = T \Leftrightarrow \forall(e : E) (in(e, S) \Leftrightarrow in(e, T)))$ 
end-spec

spec BasicSeq is
  import translate Collection by {Col → Seq, NE-Col → NE-Seq, empty-col → empty-seq,
    nonempty-col? → nonempty-seq?, insert → prepend}

  axiom conservation is  $\forall(e : E, S : Seq) \neg(empty-seq = prepend(e, S))$ 

  axiom uniqueness is
     $\forall(d : E, e : E, S : Seq, T : Seq) (prepend(d, S) = prepend(e, T) \Leftrightarrow d = e \wedge S = T)$ 
end-spec

```

Figure 10. Specs for Collection Classes

When a spec imports another spec and adds only sorts with sort axioms, operations with definitions, and theorems, the new spec is called a *definitional extension* of the old one. For example, the spec *MinMax* above is a definitional extension of *TotalOrder*. Figure 11 shows another example, which extends *BasicSet* to *Set* by adding a few more standard set operations. A definitional extension is denoted graphically by placing a *d* on or next to the morphism:

$$BasicSet \xrightarrow{d} Set$$

More generally, one spec is a definitional extension of another whenever there is a morphism between them that is one-to-one (also called injective or monic) such that the elements of the target spec that are not in the range of the morphism are all defined. Thus even if *Set* were translated, or imported into another spec and more definitions added, the result would still be a definitional extension of *BasicSet*. Translation and identity morphisms (see below) are special cases of definitional extension. Further, the composition of two definitional extensions is itself a definitional extension. The importance of the concept is that the theory generated by a spec is the same as the theory generated by any definitional extension of it. The theory already contains all possible constructed sorts, defined operations, and theorems, so a definitional extension serves only to give names to elements that in a sense are already there. The practical relevance of this observation will be seen below when the refinement constructs are presented.

Specs and morphisms together form a category called **Spec**. A *category* consists of a set or class of *objects* and a set or class of *arrows*. Each arrow has associated *domain* and *codomain* objects. For each object there is at a minimum an *identity arrow* of which it is both domain and codomain. In Slang, an *identity morphism* maps every element of a spec to itself and is indicated by a keyword:

morphism *Nat-Id* : *Nat* → *Nat* is identity-morphism

Another requirement of a category is that arrows compose associatively. If $f : A \rightarrow B$ and $g : B \rightarrow C$ are arrows then there must exist an arrow $g \circ f : A \rightarrow C$ that is their composition, and

```

spec Set is
  import BasicSet

  op delete : E, Set → Set
  definition of delete is
    axiom  $\forall (e : E) (delete(e, empty-set) = empty-set)$ 
    axiom  $\forall (e : E, S : Set) (delete(e, insert(e, S)) = delete(e, S))$ 
    axiom  $\forall (d : E, e : E, S : Set)$ 
       $(\neg(d = e) \Rightarrow delete(e, insert(d, S)) = insert(d, delete(e, S)))$ 
  end-definition

  op singleton : E → Set
  definition of singleton is
    axiom  $\forall (e : E) (singleton(e) = insert(e, empty-set))$ 
  end-definition

  op union : Set, Set → Set
  definition of union is
    axiom  $\forall (S : Set) (union(S, empty-set) = S)$ 
    axiom  $\forall (e : E, S : Set, T : Set)$ 
       $(union(S, insert(e, T)) = insert(e, union(S, T)))$ 
  end-definition
end-spec

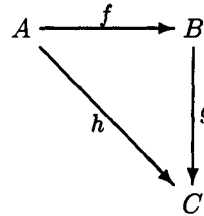
```

Figure 11. Extending *BasicSet* to *Set*

if $h : C \rightarrow D$ is also an arrow, then it must be that $h \circ (g \circ f) = (h \circ g) \circ f$. Morphism composition is ordinary function composition: to find the mapping for an element of A , first apply f and then apply g to the result. It is straightforward to prove that the result is a valid morphism.

2.1.3 Diagrams and Colimits. The *diagram* is a third fundamental concept in Slang. A diagram is a directed multigraph consisting of *nodes* and *arcs* that satisfy the requirements of a category (every node has an arc to itself, every pair of adjacent arcs implies the existence of an arc that is their composition, and composition is associative). Nodes and arcs can be assigned names, and they can also be *labeled* with the objects and arrows of some category. The names and labels are often the same, in which case only the name of the spec is shown. If a spec is used to label more than one node, however, the nodes must be given distinct names. Arcs are typically left unnamed, and identity arrows and compositions of arrows are typically assumed and not shown.

A diagram is said to *commute* if whenever there is more than one path from one node to another the paths are equal. For example, the diagram



commutes if $h = g \circ f$. The diagrams in this dissertation may be assumed to commute unless stated otherwise.

The reason that diagrams are important is that **Spec** is a category in which all diagrams possess *colimits*, and colimits provide a powerful mechanism for structuring specs hierarchically. Formally, the colimit of a diagram is an object, D , such that:

- for every object A in the diagram there is an arrow $A \rightarrow D$;
- for every arrow $f : A \rightarrow B$ in the diagram, the composition of f with $B \rightarrow D$ is equal to $A \rightarrow D$; and
- for every object E satisfying the first two conditions, there is a unique arrow $D \rightarrow E$.

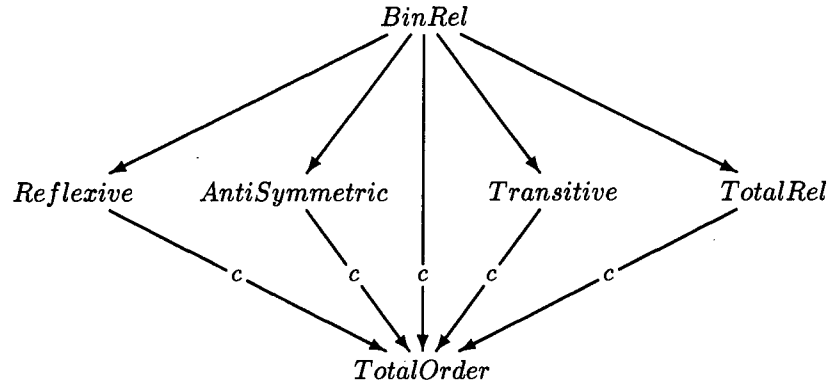
Intuitively, this means that D possesses all of the structure indicated by the diagram, and that it does so in a minimal or *universal* way in that every other object with all the structure of the diagram has at least the structure of D . A colimit is in a sense a “least upper bound” of a diagram. The set of arrows from the objects in the diagram to the colimit spec is called a *cocone*.

Diagrams and colimits allow us to build specs by composition. Figure 12 shows an alternative way to build the spec *TotalOrder* above. The spec *BinRel* is a very simple spec that defines the concept of a binary relation over a sort. The specs *Reflexive*, *AntiSymmetric*, *Transitive* and *TotalRel* define important properties of binary relations, each independently of the others. The diagram relates these separate properties by “gluing” them together on *BinRel*. The various

morphisms from *BinRel* identify all occurrences of *E* as being the same sort and the relations *br*, *rr*, *ar*, *tr*, and *tor* as being the same relation. The colimit of this diagram therefore contains one sort, called *E*, and one operation with four axioms. The name of the operation is a set containing all of the names used for it in the diagram. Any of the names in the set can be used to refer to it, but none are particularly appropriate to a total order, so the colimit is renamed. The resulting spec for *TotalOrder* is equivalent to the form given earlier. Moreover, the pieces from which it was composed can be used in different combinations to specify other kinds of relations, such as pre-orders, partial orders and equivalence relations (assuming a spec *Symmetry* is defined). Each spec defines a simple, cohesive concept, and colimits provide a rigorous means for assembling them into higher-level concepts, with a high level of reuse. Since colimits are just specs, they can be imported, translated, or used as objects in diagrams to form other colimits.

Colimits are also used to instantiate parameters. In the *Set* spec above, the sort *Set* is well-defined, with a set of constructors and other operations describing its behavior in detail. The sort *E*, on the other hand, is little more than a place holder for whatever *Set* is meant to contain. To build a specification for, say, a set of natural numbers, a diagram is used to identify the sort *E* in spec *Set* with the sort *Nat* in spec *Nat*. Figure 13 shows how this is done. The spec *Triv* contains a single sort *E* with no operations. It is used to link *E* in *Set* with *Nat* so that all three are the same sort in the colimit, *Set-of-Nat*. This structure, the colimit of a diagram consisting of two arrows with a common domain spec, is sufficiently common to have a name of its own; it is called a *pushout*.

It sometimes happens that sorts or operations in different specs have the same name but are not mapped to the same element in a colimit. To refer to such an element, a *qualified name* is used. A qualified name identifies both the name of the element and the name of the node labeled with the spec that defines the name. In the colimit *Set-of-Nat*, for example, *Triv.E* is a valid qualified name, though in this case a qualified name is not required. If the node in question is labeled with



```

spec BinRel is
  sort E
  op _ br _ : E, E → Boolean
end-spec

```

```

spec Reflexive is
  sort E
  op _ rr _ : E, E → Boolean
  axiom reflexivity is  $\forall(x : E) x rr x$ 
end-spec

```

```

spec AntiSymmetric is
  sort E
  op _ ar _ : E, E → Boolean
  axiom anti-symmetry is  $\forall(x : E, y : E) (x ar y \wedge y ar x \Rightarrow x = y)$ 
end-spec

```

```

spec Transitive is
  sort E
  op _ tr _ : E, E → Boolean
  axiom transitivity is  $\forall(x : E, y : E, z : E) (x tr y \wedge y tr z \Rightarrow x tr z)$ 
end-spec

```

```

spec TotalRel is
  sort E
  op _ tor _ : E, E → Boolean
  axiom totality is  $\forall(x : E, y : E) (x tor y \vee y tor x)$ 
end-spec

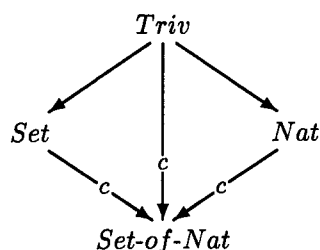
```

```

spec TotalOrder is
  translate colimit of diagram
    nodes BinRel, Reflexive, AntiSymmetric, Transitive, TotalRel
    arcs BinRel → Reflexive : {br → rr}, BinRel → AntiSymmetric : {br → ar},
        BinRel → Transitive : {br → tr}, BinRel → TotalRel : {br → tor}
  end-diagram by {br → ≤}

```

Figure 12. Properties of Binary Relations



spec *Triv* **is** sort *E* **end-spec**

spec *Set-of-Nat* **is**

colimit of diagram

nodes *Triv, Set, Nat*

arcs *Triv* → *Set* : {}, *Triv* → *Nat* : {*E* → *Nat*}

end-diagram

Figure 13. Spec for a Set of Natural Numbers

a colimit spec, it may be necessary to use two or more levels of qualification to generate a unique identifier.

The colimit operation creates not only the colimit object but also the cocone morphisms. These morphisms can be referred to using a keyword, as in

morphism *Set-to-Set-of-Nat* : *Set* → *Set-of-Nat* **is cocone-morphism from** *Set*

2.1.4 A Final Example. Figures 14 and 15 illustrate how a complex spec can be built up gradually from small pieces in a well-structured way. The final spec, *PropLogic*, is a domain theory for a subset of propositional logic. It contains a sort *Formula* that can be used to represent and manipulate propositional formulas in conjunctive normal form (or disjunctive normal form, for that matter: the structure does not impose either interpretation uniquely). The spec is built up by first defining what propositional variables and terms are, then constructing sets of terms as clauses and sets of clauses as formulas (the terminology chosen reflects the preferred interpretation). It also instantiates *FiniteMap* as an assignment from variables to boolean values. Each colimit is renamed to collapse equivalence classes of names to the preferred name, and set operations from

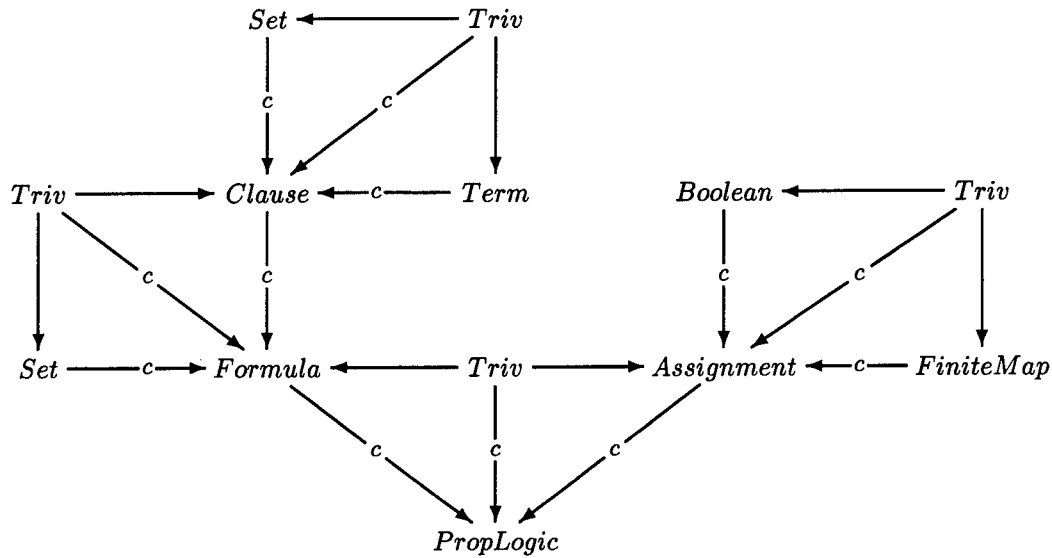


Figure 14. Diagram for Propositional Logic Theory

the two copies of *Set* are renamed to avoid having to use qualified names extensively, in morphisms or in specs that import *PropLogic*. This domain theory will be used in Chapter IV to specify the boolean satisfiability problem.

Note that in Figure 14 there are several nodes with the same name, because they are labeled with the same spec. This is not an error since the figure is just a picture of several diagrams shown together for illustrative purposes. No diagram used to compute a colimit has two nodes of the same name, even though the names are reused from diagram to diagram.

2.2 Refinement Constructs

The refinement constructs of Slang build on the specification constructs described above. Refinement allows the user to modify the description of a system in a formal way to change its level of abstraction without changing its meaning.

2.2.1 Interpretations. The fundamental unit of refinement is the *interpretation*. An interpretation is a diagram of the form

```

spec Term is
  sorts P, T

  op pos : P → T
  op neg : P → T
  op var-of : T → P
  op val-of : T → Boolean

  constructors {pos, neg} construct T

  definition of var-of is
    axiom  $\forall(p : P) (var-of(pos(p)) = p)$ 
    axiom  $\forall(p : P) (var-of(neg(p)) = p)$ 
  end-definition

  definition of val-of is
    axiom  $\forall(p : P) (val-of(pos(p)) = true)$ 
    axiom  $\forall(p : P) (val-of(neg(p)) = false)$ 
  end-definition
end-spec

spec Clause is
  translate colimit of diagram
    nodes Triv, Set, Term
    arcs Triv → Set : {}, Triv → Term : {E → T}
  end-diagram by
    {E → T, Set → Clause, NE-Set → NE-Clause, delete → c-delete,
     empty-set → empty-clause, nonempty-set? → nonempty-clause?, in → c-in,
     insert → c-insert, singleton → c-singleton, union → c-union}

spec Formula is
  translate colimit of diagram
    nodes Triv, Set, Clause
    arcs Triv → Set : {}, Triv → Clause : {E → Clause}
  end-diagram by
    {E → Clause, Set → Formula, NE-Set → NE-Formula, delete → f-delete,
     empty-set → empty-formula, nonempty-set? → nonempty-formula?, in → f-in,
     insert → f-insert, singleton → f-singleton, union → f-union}

spec Assignment is
  translate colimit of diagram
    nodes Triv, FiniteMap, Boolean
    arcs Triv → Boolean : {E → Boolean}, Triv → FiniteMap : {E → Cod}
  end-diagram by {Map → Assignment}

spec PropLogic is
  translate colimit of diagram
    nodes Triv, Assignment, Formula
    arcs Triv → Formula : {E → P}, Triv → Assignment : {E → Dom}
  end-diagram by {E → P}

```

Figure 15. Building a Theory of Propositional Logic

$$A \longrightarrow A\text{-as-}B \longleftarrow d \longrightarrow B$$

that is, two morphisms with a common codomain. The middle spec, called the *mediator*, shows how the components of the *source* spec on the left can be defined in terms of the components of the *target* spec on the right. The mediator is a definitional extension of the target, so nothing has been added to the target that was not already present in the theory it generates. A shorthand notation for an interpretation is $A \Rightarrow B$, hiding the mediator.

Figure 16 shows how sets can be refined to sequences. First an equivalence relation, *perm*, is defined on sequences. This relation is analogous to the equality theorem for sets. That is, two sequences are considered permutations of each other if they contain the same elements, regardless of order or multiplicity (this is not the standard definition of permutation of sequences). The sort *Set-as-Seq* is then defined as the quotient sort *Seq/perm*. Finally, set operations are defined for this quotient sort in terms of the underlying sequence operations.

An *interpretation scheme* is also a pair of morphisms with a common codomain, but neither is required to be a definitional extension. This can be thought of as an interpretation with “holes” in it, or an interpretation that is not complete. Conversely, an interpretation can be viewed as a special case of an interpretation scheme. The Slang syntax for an interpretation scheme is the same as that for an interpretation, except the keyword “interpretation” is replaced with the keyword “ip-scheme”.

Two interpretations $A \Rightarrow B$ and $B \Rightarrow C$ can be composed as shown in Figure 17. The mediator of the composition is formed as a colimit (a pushout, in fact), and the domain and codomain morphisms are formed by composition. An important property of colimits is that they preserve definitional extension: if some morphisms in a diagram are definitional extensions, then certain corresponding cocone morphisms of the colimit will be as well. In the figure, the cocone morphism $B\text{-as-}C \rightarrow A\text{-as-}C$ is definitional because $B \rightarrow A\text{-as-}B$ is definitional. The former is then composed with $C \rightarrow B\text{-as-}C$ to yield $C \rightarrow A\text{-as-}C$, also definitional. Interpretation schemes

$$\text{BasicSet} \longrightarrow \text{Set-as-Seq} \xleftarrow{d} \text{BasicSeq}$$

```

spec Set-as-Seq is
  import BasicSeq

  op in-seq : E, Seq → Boolean
  definition of in-seq is
    axiom ∀(e : E) ¬in-seq(e, empty-seq)
    axiom ∀(d : E, e : E, S : Seq) (in-seq(e, prepend(d, S)) ⇔ e = d ∨ in-seq(e, S))
  end-definition

  op perm : Seq, Seq → Boolean
  definition of perm is
    axiom ∀(S : Seq, T : Seq) (perm(S, T) ⇔ ∀(e : E) (in-seq(e, S) ⇔ in-seq(e, T)))
  end-definition

  sort Set-as-Seq
  sort-axiom Set-as-Seq = Seq/perm

  op empty-set : Set-as-Seq
  definition of empty-set is
    axiom empty-set = (quotient perm)(empty-seq)
  end-definition

  sort NE-Set
  sort-axiom NE-Set = Set-as-Seq | nonempty-set?

  op nonempty-set? : Set-as-Seq → Boolean
  definition of nonempty-set? is
    axiom ∀(S : Set-as-Seq) (nonempty-set?(S) ⇔ ¬(S = empty-set))
  end-definition

  op in-set : E, Set-as-Seq → Boolean
  definition of in-set is
    axiom ∀(e : E, S : Seq) (in-set(e, (quotient perm)(S)) = in-seq(e, S))
  end-definition

  op insert : E, Set-as-Seq → Set-as-Seq
  definition of insert is
    axiom ∀(e : E, S : Seq)
      (insert(e, (quotient perm)(S)) = (quotient perm)(prepend(e, S)))
  end-definition
end-spec

interpretation Set-to-Seq : BasicSet ⇒ BasicSeq is
  mediator Set-as-Seq
  domain-to-mediator {Set → Set-as-Seq, in → in-set}
  codomain-to-mediator import-morphism

```

Figure 16. Refining Sets to Sequences

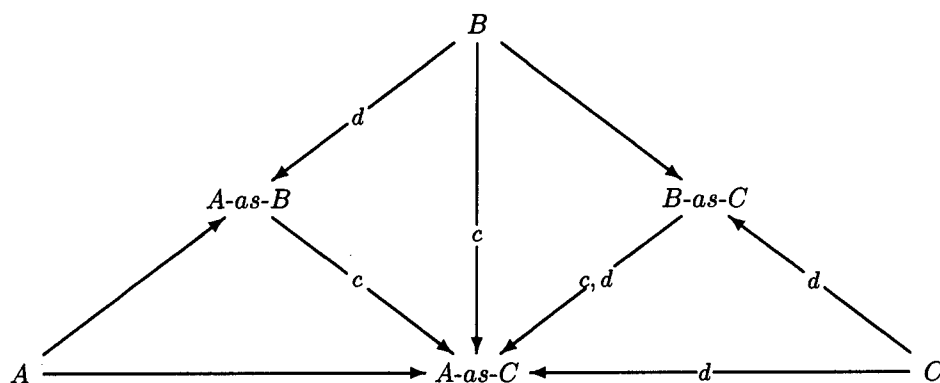


Figure 17. Composition of Interpretations

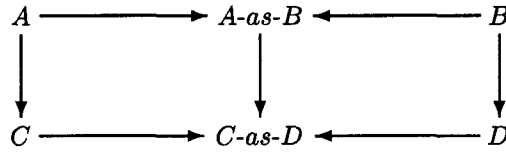
are composed in the same way to yield new interpretation schemes, and in fact the two can be mixed. Composition of interpretations or interpretation schemes is associative, and every spec has an identity interpretation (scheme) with itself as source, mediator and target related by identity morphisms, so interpretation schemes constitute the arrows of a second category over specs, with interpretations proper forming a subcategory. However, diagrams in these categories do not, in general, have colimits.

An interpretation refines a spec by translating the vocabulary of one level of abstraction to the vocabulary of a lower one. The target spec—or rather, the theory it generates—has at least the structure of the source and possibly more. The additional structure is added detail that moves the specification toward implementation. Refinement is the process of chaining together a series of interpretations until the resulting target spec is directly implementable in some target language. The details of how SPECWARE treats target languages algebraically and how code generation is done are not relevant to the research to be described and so will be omitted; the interested reader is referred to (80).

Morphisms also represent a kind of refinement, but are not as general since the target spec must contain all the necessary structure without benefit of extension. Interpretations therefore often exist between specs where morphisms do not. When a uniform treatment is desired, a morphism

can be *raised* to an interpretation by using two copies of the target spec, one to be the mediator and one the target of the interpretation, linked by an identity morphism. A morphism and an interpretation can be composed with each other, for example, by first raising the morphism.

2.2.2 Interpretation Morphisms. An interpretation or interpretation scheme can be treated as an object and a notion of arrow defined for it. An *interpretation scheme morphism* is a triple of morphisms from the source, mediator and target specs of one interpretation scheme to the corresponding specs of another,



such that the resulting diagram commutes. Interpretation scheme morphisms can be composed by composing their component morphisms; this composition is associative. Every interpretation scheme has an interpretation scheme morphism to itself composed of three identity morphisms. Thus interpretation schemes and their morphisms constitute a category, called **Interp**, with interpretations and their morphisms as a subcategory.

The composition just described is called *sequential composition*. Interpretation scheme morphisms can also be composed in parallel. *Parallel composition* is defined when the codomain morphism of one interpretation scheme morphism is the same as the domain morphism of another, as shown in Figure 18 with $B \rightarrow E$ as the common morphism. The composition is computed by composing the top pair of interpretation schemes, $A \Rightarrow B$ with $B \Rightarrow C$, and the bottom pair, $D \Rightarrow E$ with $E \Rightarrow F$, each by computing a colimit and composing morphisms. There is a unique morphism $A-as-C \rightarrow D-as-F$ that makes the diagram commute. This morphism is the mediator morphism of the composed interpretation scheme morphism from $A \Rightarrow C$ to $D \Rightarrow F$.

2.2.3 Diagram Refinement. Diagrams can be defined for the category **Interp** just as they were for **Spec**, and like them, diagrams in **Interp** have colimits. This fact forms the basis

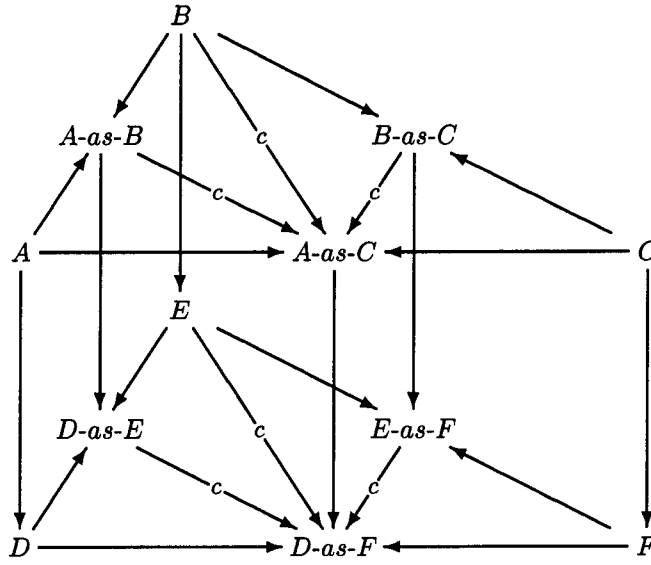


Figure 18. Parallel Composition of Interpretation Scheme Morphisms

of a technique of incremental refinement called *diagram refinement*. If a spec is constructed as a colimit, it is natural to want to take advantage of that structure when constructing a refinement of it. Figure 19 shows how to take the spec for a set of natural numbers constructed above and refine it to a sequence of natural numbers. The spec *Set-of-Nat* is a colimit. To each node and arc of the corresponding diagram is attached an interpretation and an interpretation morphism, respectively. In this case, only the spec *Set* is being refined non-trivially, using the interpretation to *Seq* presented earlier; the other specs are refined with identity interpretations, and the interpretation morphisms are the obvious ones. The colimit of a diagram of interpretations (or schemes) is computed by forming the respective colimit specs and then finding the unique arrows between them that make the resulting diagram commute.

The formal definition of diagram refinement is shown graphically in Figure 20. I_1 and I_2 are *shapes*, meaning graphs whose nodes and edges are not labeled. The arrow d_1 is a diagram in **Spec**. Formally it is a functor (54) from the category I_1 to **Spec**, showing the specs and morphisms with which the nodes and arcs of I_1 are labeled. The arrow δ is a diagram in **Interp**. It has the same shape as d_1 , but labels the nodes and arcs with objects and arrows from a different category. The

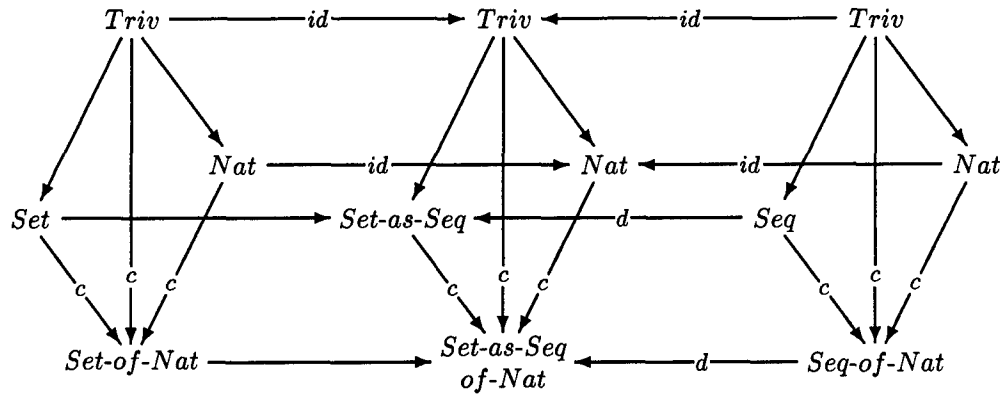


Figure 19. Example of Diagram Refinement

arrow *dom* is a functor from **Interp** to **Spec**. It maps an interpretation to its domain spec and an interpretation morphism to its domain morphism. Figure 20 commutes, which means the diagram δ must label I_1 with refinements whose domains are the specs and morphisms of d_1 . The arrow *cod* is another functor from **Interp** to **Spec**; this one maps interpretations to their codomain specs and interpretation morphisms to their codomain morphisms. The arrow from I_1 to the bottom right copy of **Spec** is the composition of δ with *cod*. This much of the diagram corresponds to the example: the **Spec** diagram defining *Set-of-Nat* is d_1 , the **Interp** diagram of refinements is δ , and the two diagrams agree on which specs and morphisms are being refined.

The arrow σ is a *shape morphism* from I_1 to a second shape, I_2 . It maps each node and arc in I_1 to a node or arc in I_2 , but it may map more than one node or arc in I_1 to the same node or arc in I_2 , or I_2 may have nodes and arcs that I_1 does not. This device allows the shape of the diagram to change from source spec to target spec. The d_2 arrow labels I_2 with specs and morphisms, and commutativity with *cod* implies that the codomain specs and morphisms of δ are all mapped to themselves. That is, we are not changing the specs or morphisms of the codomain diagram, but only its shape. If the codomain diagram contains two nodes labeled with the same spec, these nodes might be identified in I_2 . Similarly, if we wish to identify a shared part between two codomain specs, sharing that did not exist between the corresponding domain specs, we can

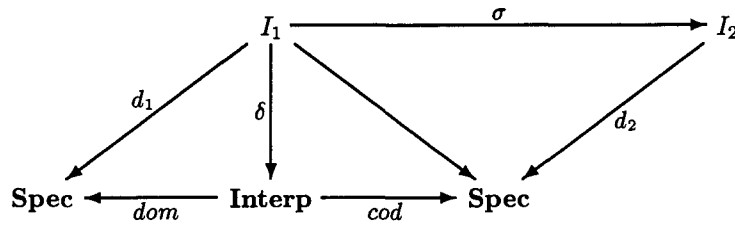


Figure 20. Diagram Refinement

add a node in I_2 with arcs to the nodes that are to share it. We will see an example of a shape morphism in Chapter V.

If all of the refinements in a diagram refinement are interpretations, then the colimit refinement will also be an interpretation. Sometimes the colimit is an interpretation even when some of the component refinements are *not* interpretations. This happens when the “holes” that exist in one refinement occur in a shared part and are “filled in” by some other refinement. For example, an operation that in one refinement has no definition can become identified in a colimit with an operator in another refinement that does have a definition. Interpretation schemes thus play an important role in refinement, even though refinement is not considered complete until only interpretations remain.

III. Characterizing Local Search Algorithms

Local search algorithms search the space of feasible solutions by generating "new" solutions from existing ones, seeking improvement in the cost of the solutions generated. "New" is in quotes because local search is not systematic, in that there is nothing inherent in the approach that prevents a solution from being generated many times, or that guarantees that every feasible solution will eventually be generated or somehow ruled out. Local search is for the most part memoryless: no record is maintained of which solutions have been generated or what their costs were. This helps to keep the resource requirements (e.g., memory and CPU time) of local search algorithms modest and permits them to be used on problem instances of great size.

Local search is in general not *exact*, which means that some local search algorithms do not guarantee that the solution found is optimal. Local search is more often used for semi-optimization problems or even without any guarantees regarding optimality, relying instead merely on a tendency to produce "good" solutions. Local search can also provide a cheap means to improve a solution found by some other means, such as a heuristic greedy algorithm (19).

These characteristics distinguish local search algorithms from global search algorithms, which in general can be made exact. Applied to optimization, global search searches a space of feasible solutions to find one or more optimal ones. Exactness is achieved by searching systematically. The process of beginning with an initial space and splitting, extracting and pruning until nothing remains to be split means that every feasible solution is at least indirectly considered, a technique known as implicit enumeration. The solutions found are thus guaranteed to be optimal. The downside of global search is that search spaces are often very large, and the number of subsets of solutions is even larger, so that even when sophisticated techniques are used to restrict subsets or eliminate them from consideration, global search in general requires large amounts of memory and CPU time. It is thus practical to use it only on relatively small problem instances, or on well-behaved problems (i.e., not NP-complete ones).

An example will highlight the importance of these differences. Bart Selman and his colleagues have published some comparisons between different approaches to the boolean satisfiability problem (64). The standard way to solve this problem exactly is with the Davis-Putnam algorithm, an instance of global search. A MIPS workstation running a good implementation of this algorithm can solve a randomly generated "hard" formula with 140 variables, 3 terms per clause and 602 clauses in about 4.7 hours of CPU time. Given the exponential growth of the search space, the estimated time to solve a formula with 150 variables is several days. By reformulating the problem as minimizing the number of unsatisfied clauses, a simple local search algorithm can be used. This algorithm was consistently able to find satisfying assignments for formulas that had them, with up to 500 variables and 2,150 clauses, in just over 1.6 hours and with a much more modest rate of increase in CPU time with increasing problem instance size. Two observations are key here. First, when the Davis-Putnam algorithm finds no satisfying assignments, it is certain that there are none, whereas when the local search algorithm finds no satisfying assignments there is no such guarantee. Second, for seemingly reasonable problem instances, Davis-Putnam is useless while the local search algorithm can produce answers that may be quite useful. The speed and power of local search and its applicability to a great many problems of practical interest make it a very important class of algorithms.

There is tremendous variety in local search algorithms. Many different approaches have been taken to address the problems of avoiding regeneration of solutions and finding the best solutions. This variety is another major reason why local search is interesting and important. We seek to formalize a model broad enough to encompass most or all of local search and then to specialize it to specific approaches. First, then, we will describe the characteristic features of local search in an informal setting and outline some of the ways these features are realized in particular algorithms. The common features can be grouped into three categories: overall strategy, finding an initial solution, and search strategy.

3.1 Overall Strategy

The overall strategy adopted by a local search algorithm helps to determine the character of many of the other components and acts to coordinate their activities. One aspect of strategy is how many trials are run. Since local search does not guarantee optimality in all cases, running multiple trials from different starting points can produce answers of different quality, from which the best can be chosen. Trials can be run sequentially or in parallel; when parallel, solutions might "compete" with each other such that new solutions tend to be generated from the better current ones. This serves to intensify the search near promising regions of the solution space.

A second aspect of strategy concerns how the solution space is defined and searched. In many cases the space of feasible solutions is embedded in a larger space that includes solutions of the same general form that violate one or more of the conditions of the problem. It is sometimes valuable to include some or all of the infeasible solutions in the search. For example, some problems have special structure that allows the optimality of a solution to be determined by a simple test. One search technique in this case is to find an initial solution that satisfies this condition but may be infeasible, then look at which constraints are violated and incrementally change the solution to satisfy them. When all constraints are met, the optimal solution is at hand. In linear programming this approach is called a *dual* method. More generally, methods that search within the feasible region are called *internal* methods and those that search from without are called *external* methods. It is possible to combine both techniques, either by alternating periods of search in each region or by simply searching the larger space without considering whether any given solution is feasible or infeasible until the search is terminated.

A third aspect of strategy is whether or not to collect some sort of data during search. As stated above, local search is memoryless in that it keeps no detailed record of where it has been. It can be useful, however, to gather and use statistical data about the solutions visited. This data can be used either to intensify search in regions that seem promising, or conversely to diversify the

search by forcing subsequent trials to begin at initial solutions that are unlike the solutions already searched.

3.2 Finding an Initial Solution

A critical element of local search is finding an initial solution from which to launch the search. The choice of initial solution can have great impact on both the speed with which the search converges to a final solution and the quality of that solution. Aspects of the overall strategy also influence what sort of initial solutions are needed. For example, if multiple trials are to be run, a variety of initial solutions are needed, whereas if only one trial is to be run, only one initial solution is needed.

Finding an initial solution to a problem with difficult constraints can be challenging. In general, any algorithmic approach can be applied to this subtask. Within the bounds set by the overall strategy, this subtask is largely independent of the other elements of a local search algorithm.

There are two classes of methods for finding an initial solution, each with two subclasses. The first class is *uninformed* methods. These make no attempt to find a “good” solution or to resemble or avoid solutions already searched. If only one initial solution is needed, an *arbitrary* choice is made. This involves constructing or searching for a feasible solution by some means that always produces the same answer for any given input. When multiple initial solutions are needed, a *random* choice can be made. That is, there is some element of randomness to how the solution is constructed or found, so that for any given input there is a set of possible outcomes. One typically prefers a uniform distribution of initial solutions that samples the entire solution space to provide a simple diversification strategy, but rarely is this quality demonstrated formally.

The second class of methods is *informed* methods. These take some advantage of additional information beyond the need for feasibility. A *heuristic* choice attempts to produce an initial solution with above-average quality, in the hope that the final solution will either be of high

quality, be found quicker, or both. Alternatively, a *novel* choice uses data gathered in previous trials to diversify the search deliberately rather than randomly. Both heuristic and novel choices may contain a random element as well.

3.3 Search Strategy

Once an initial solution (or a set of them, as required by the overall strategy) has been found, search proceeds by generating new solutions until some stopping criterion is met. At this point the best solution seen is returned, possibly along with other information about the search, with further actions as dictated by the overall strategy. Search proceeds in steps or moves. Each step is defined by a state consisting of the current solution(s) and possibly other information. The search strategy defines a function on these states, describing how new states are generated, and which states are final. As stated above, only the current state is maintained; no record is kept of prior states. The search strategy consists of three components: neighborhood structure, move selection rules, and stopping criteria. Neighborhood structure describes the topology of the solution space, that is, which solutions are generable from which other solutions. Move selection rules define which of the generable solutions will be in the successor state, and updates other state information as necessary. Stopping criteria determine when the search halts. The following subsections describe these components in more detail.

3.3.1 Neighborhood Structure. Neighborhood structure fundamentally defines what moves are possible by defining how new solutions are generated from old ones. In the simplest and most common case, the neighborhood structure is described by a binary relation on solutions. This relation defines for each solution a set of neighboring solutions. Typically, the relation is such that neighboring solutions are highly similar. Intuitively, such a neighborhood is generated by the action of a function that perturbs or incrementally transforms the solution. More generally, a neighborhood structure can be a higher-order relation describing how multiple solutions can be

combined to produce new ones. For example, the crossover operators used in genetic algorithms define ternary relations that describe all the ways that two solutions can be crossed to produce offspring.

Properties of relations (symmetry, reflexivity, etc.) indicate structure in the neighborhood that can potentially be exploited by local search algorithms. One particularly important property of binary relations is *reachability*, which means that from any feasible solution it is possible to reach any other through a sequence of neighboring solutions. If the neighborhood relation is viewed as a directed graph, where the nodes are solutions and edges connect neighboring solutions, then reachability is equivalent to saying that this graph is strongly connected. Reachability is not required for local search, but clearly it is a nice property to be able in principle to search the entire solution space from any initial solution. Another useful property is *feasibility*, which states that all neighbors of feasible solutions are feasible. This property is sufficient, but not necessary, to insure that the solution returned by the search is a feasible one. If a neighborhood is both reachable and feasible, we call it a *perfect* neighborhood. Most typical examples of local search are over perfect neighborhoods. Higher-order relations can be treated as hypergraphs and corresponding definitions of reachability and feasibility are possible, but this is not critical since algorithms that use higher-order relations usually use them in conjunction with a binary "mutation" relation anyway.

The relationships between the neighborhood structure and the cost function are also important. If the cost of a solution is no worse than that of all of its neighbors, it is called a *local optimum*. If all local optima are global optima, then the neighborhood is called *exact* with respect to the cost function. The nature of local search is that it seeks local optima, so when exactness is not satisfied the search must somehow "escape" local optima and continue searching. Neighborhoods that are "smooth" are generally easier to search than "spiky" ones in which neighboring solutions often differ radically in cost. Sometimes it happens that all local optima are feasible for neighborhoods

that are not feasible. In this case the search can range freely over infeasible solutions because the act of seeking local optima guarantees feasibility of the solution returned.

The size of a neighborhood is another important factor when designing an algorithm. If a neighborhood is exact and each solution has only a small number of neighbors, then search can be done very quickly and optimal solutions found. When this is not the case, there is a design tradeoff between search speed and solution quality. An inexact neighborhood can always be enlarged until it is exact, if by no other means than by making all solutions neighbors. (As might be expected, NP-complete problems such as the traveling salesman problem require neighborhoods that grow exponentially with problem instance size in order to remain exact.) Large neighborhoods are slow to search, but if they are "almost" exact or otherwise well-behaved, good solutions will be found. Defining a small neighborhood means it can be searched quickly but doing so may lead to proliferation of local optima. On the other hand, if the drop in average quality is not too extreme it may be more effective, given a fixed amount of time, to search quickly many times and sample the search space widely than to search slowly only one or a few times.

3.3.2 Move Selection Rules. The neighborhood structure defines the set of possible moves, but not all moves are created equal, so the search strategy includes a set of selection rules that are used to evaluate and select which move is accepted at each step. The selection rules together with the underlying neighborhood structure determine the quality of solution found and the probability that the search will get trapped in a loop of recurring states or otherwise fail to make progress. Some search strategies guarantee that search states will not recur, others offer a low probability of recurrence, and still others make no promises whatsoever.

Moves can be classified first as improving, neutral, or worsening, according to whether the solutions they generate are better, the same, or worse, respectively, than the solutions that generate them. Clearly improving moves are preferred, but it is sometimes necessary to accept neutral or worsening moves in order to find better solutions that are more than one move away. Neutral

moves, for example, can be used to escape "weak" local optima or *plateaus*, where no neighbors of a solution are better but a better solution is reachable by a sequence of moves through solutions of equal cost. In Selman's work on boolean satisfiability there was a dramatic difference in the ability of the algorithm to find satisfying assignments depending on whether neutral moves were accepted or not (64).

Selection rules evaluate moves in these categories and decide which to accept. The simplest and most obvious rule is to accept any improving move that is available. This is called hill climbing, and is generally considered the essence of local search. Steepest ascent is a refinement of hill climbing that always selects the move that makes the greatest improvement. This may lead to a shorter search, but also requires spending more time at each step.

A second rule is to forbid certain moves. For example, to escape a local optimum it may be necessary to accept several worsening moves. After the first is made, there is an improving move available that returns the search to the local optimum. By forbidding moves that undo moves recently made, the search may be taken far enough from the local optimum to make progress again without revisiting states. Forbidden moves can also be used to control alternating phases of interior and exterior search. Declaring moves to be forbidden is the defining characteristic of tabu search (23), described further below.

A third rule is to accept moves stochastically based on their relative cost. For example, random noise can be added to the cost function before classifying the move as improving, neutral, or worsening. Thus some improving moves will be rejected and some worsening moves accepted. By controlling the amplitude of the noise and reducing it over time, the search can explore a large portion of the solution space before converging on a local optimum. This approach is the defining characteristic of simulated annealing (40). Genetic algorithms also accept moves stochastically (46), though not in quite the same way as in simulated annealing.

A fourth rule is to evaluate solutions more than one move away from the current one. Moves are evaluated out to some fixed or bounded but variable depth. Based on this evaluation, a sequence of moves is selected. Typically the sequence with the best net effect is the one chosen, and it may include improving, neutral, and worsening moves. The new state is the final state of the sequence. This technique is one way to search a larger neighborhood, in effect, without changing the definition of the neighborhood relation. It also often implicitly includes a form of forbidden moves, in that each move in the sequence removes some moves from consideration later in the sequence. Once a sequence of moves is accepted, however, all such restrictions are dropped.

A final rule is to accept moves either arbitrarily or randomly. This is a weak selection rule and is usually used only for tie breaking. Most local search algorithms use a hierarchy of rules, such as choose the best available move that is not forbidden and break ties randomly. Random tie breaking reduces the chances of states recurring, if the problem is such that recurrence is possible. Sometimes tie breaking can be done systematically in such a way that recurrences are prevented completely, for example by indexing variables and always choosing the lowest index (59).

Selecting a move is a significant subproblem within local search, and like finding an initial solution, it has been solved using a wide variety of algorithmic approaches. The task of finding the move that yields the greatest improvement in the cost function, for example, is an optimization problem in its own right. It can be solved simply by enumerating the entire neighborhood, by implicit enumeration such as global search, or even by local search. Alternatively, the neighborhood may be sampled at each step by some heuristic that generates only the most promising neighbors for further consideration, as in *candidate list* strategies (23).

3.3.3 Stopping Criteria. The third and final element of the search strategy is deciding when to stop searching. Stopping criteria can be based on the solutions that have been seen during the search or on other aspects of the search state.

The most obvious stopping criterion is to stop when a globally optimal solution has been found. When searching an exact neighborhood, a necessary and sufficient condition for global optimality is local optimality. Even with an inexact neighborhood, however, sufficient conditions may be known and can be used to terminate the search. For example, the cost function may have known bounds, so if a solution at the bound is found then it is known to be globally optimal. When conditions equivalent to global optimality are not known or are impractical to use, additional stopping criteria are needed.

Another common stopping rule is to stop when no improving moves are available, that is, at a local optimum, even when the solution might not be globally optimal. There are many ways to try to escape from a local optimum, and certainly one way is to stop searching and start over somewhere else.

Several of the selection rules described above are intended to enable the search to be continued even after a local optimum has been reached. Most of these offer no clear guidance on when to stop, so resource limits are imposed on the search. Two common criteria are to stop after a fixed number of moves have been made either since the search began or since the best solution seen in the search was found.

Another rule is to use a threshold on the probability that a solution better than the current best can ever be found. This is the kind of rule typically used in simulated annealing. As the amplitude of the cost function noise drops, the chances of accepting a worsening move decrease. Once a local optimum is reached, no improving moves are available. Search stops after the observed ratio of accepted to non-accepted moves drops below a threshold.

Another class of rules is to stop when the solution found is good enough. As before, if bounds on the cost function are known, then precise limits on the degree of suboptimality that is acceptable can be used. Some problems have thresholds on the solution cost itself that define when a solution will do.

Finally, a rule that conceptually blends many of the above ideas is to stop when the rate of improvement in solution cost drops below some threshold, under the assumption that significant further improvement is unlikely.

3.4 Examples of Local Search Algorithms

The previous sections have described the common elements of local search and a large number of variants among these elements. This section presents several example local search algorithms in order to show how the elements are combined into a coherent algorithm and how the nature of the problem being solved influences the design choices made. Some of the algorithms are shown applied to particular problems, while others are generic and represent classes of algorithms. Algorithms are presented in an indeterminate high-level language, with English annotations when convenient.

The simplex algorithm for solving linear programs is a classic local search algorithm (5, 59).

A linear program in standard form,

$$\min \mathbf{c}\mathbf{x} \text{ subject to } \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$$

is a continuous optimization problem, in that the variables are real-valued, but one can show analytically that optimal solutions always occur at extreme points of the feasible region defined by the constraints, or on edges or faces connecting optimal extreme points. Thus the number of solutions that actually need to be considered is finite. The extreme points in turn correspond to bases of the constraint matrix, $\mathbf{A}$, so the problem reduces to finding a subset of the variables, called *basic variables*, that yields an optimal solution. Figure 21 shows the simplex algorithm written at a relatively high level of abstraction. The neighborhood structure is defined by the operation of choosing a non-basic variable x_k to enter the basis and a basic variable x_r to leave, with checks that the new basis is feasible and has a solution. This neighborhood is exact, so the overall strategy is to search once, keeping track of one current solution, with no other information needing to be gathered. The move selection rule is to select the direction of steepest improvement, for which an

efficient test exists (the max expression), and move in that direction to the adjacent extreme point (the min expression). In a so-called degenerate solution, more than one basis corresponds to the same extreme point. In this case, a move will in fact be a neutral move. The possibility exists in this situation that the search will repeatedly return to the same solution, so some implementations use indexing schemes to guarantee that a degenerate extreme point will eventually be escaped. In the absence of degeneracy, all moves are improving moves and the search stops at a local optimum, which by exactness is guaranteed to be a global optimum. There is an efficient test for optimality, too, so enumerating the neighborhood of a solution explicitly is never needed. An arbitrary initial solution to start the search is typically found by solving a relaxed version of the problem that has no constraints and a cost function that measures the degree to which a solution violates the constraints. This subproblem is solved using the simplex algorithm from a standard initial solution. The so-called dual simplex algorithm reverses the roles of the feasibility and optimality conditions: an initial solution is found that satisfies the optimality condition but is infeasible, and new solutions are generated to reduce the infeasibility until a feasible (and optimal) solution is found. Finely honed implementations of the simplex algorithm exist that can solve very large problems in a reasonable amount of time. The worst-case performance of the simplex algorithm is exponential in the number of variables and constraints, but such performance is extremely rare and the average performance is linear.

Selman et al. (64) apply a less elaborate form of local search to the boolean satisfiability problem. Figure 22 shows their algorithm, which they call GSAT. The input is a boolean formula in conjunctive normal form, and the output is an assignment of variables to boolean values that minimizes the number of unsatisfied clauses. They define a feasible assignment to be any assignment whatsoever over the variables of the formula. Initial assignments are generated by randomly assigning each variable a value. The neighbors of a solution are those that differ in the value assigned to exactly one variable. At each move, all neighboring assignments are generated, their costs calculated, and the best move is made, with ties broken randomly to prevent getting stuck

```

procedure Simplex
Input: an  $n$ -vector  $\mathbf{c}$  of cost coefficients, an  $m \times n$  matrix  $\mathbf{A}$  of
      constraint coefficients, and an  $m$ -vector  $\mathbf{b}$ 
Output: an  $n$ -vector  $\mathbf{x}$  containing the optimal solution to
      Minimize  $\mathbf{c}\mathbf{x}$ 
      Subject to  $\mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ 
      or an indication that the problem is infeasible or unbounded
begin
  Find an initial basic feasible solution to  $A\mathbf{x} = \mathbf{b}$  with basis  $B$ ,
  or return "problem is infeasible" if none exist
  loop
     $\bar{\mathbf{b}} := B^{-1}\mathbf{b}$ 
     $\mathbf{w} := \mathbf{c}_B B^{-1}$ , where  $\mathbf{c}_B$  is the cost coefficient vector of the basic variables
     $k := \text{some}(k) \mathbf{w}a_k - c_k = \max(\{\mathbf{w}a_j - c_j \mid (j) j \in R\})$ , where  $R$  is the set
      of indices associated with the non-basic variables
    exit when  $\mathbf{w}a_k - c_k \leq 0$ 
     $\mathbf{y}_k := B^{-1}a_k$ , where  $a_k$  is the  $k$ -th column of  $A$ 
    if  $y_k \leq 0$  then
      return "Optimal solution is unbounded"
    end if
     $r := \text{some}(r) \frac{\bar{b}_r}{y_{rk}} = \min(\{\frac{\bar{b}_i}{y_{ik}} \mid (i) 1 \leq i \leq m \wedge y_{ik} > 0\})$ 
     $B := B$  with  $a_k$  less  $a_r$ 
  end loop
  return basic variables  $x_B = \bar{\mathbf{b}}$ , non-basic variables  $x_N = 0$ ,
  and  $z = \mathbf{c}_B \mathbf{x}_B$  as the optimal solution
end procedure

```

Figure 21. Simplex Algorithm for Linear Programs

```

procedure GSAT
Input: a set of clauses  $\alpha$ , MAX-MOVES, and MAX-TRIES
Output: a satisfying truth assignment of  $\alpha$ , if found
begin
  for  $i := 1$  to MAX-TRIES loop
     $T :=$  a randomly generated truth assignment
    for  $j := 1$  to MAX-MOVES loop
      if  $T$  satisfies  $\alpha$  then return  $T$ 
       $p :=$  a randomly chosen propositional variable such that a change
        in its truth assignment gives the largest increase in the total
        number of clauses of  $\alpha$  that are satisfied by  $T$ 
       $T := T$  with the truth assignment of  $p$  reversed
    end loop
  end loop
  return "no satisfying assignment found"
end procedure

```

Figure 22. GSAT Algorithm for Boolean Satisfiability

on a plateau (although it may still take a very long time to escape a big one!). The authors claim that worsening moves are rarely chosen and so are ignored. The cost of a solution is defined to be the number of unsatisfied clauses in the formula. Search terminates when a solution with a cost of zero is found, since this is obviously sufficient for global optimality, or if no improving or neutral move is available, or if the maximum number of search steps has been reached. The neighborhood is not exact for the cost function, so the whole process is repeated some number of times, unless a satisfying solution has been found. As mentioned above, this algorithm is very good at finding solutions to satisfiable formulas much larger than exact algorithms such as Davis-Putnam can solve, but cannot prove that a formula is unsatisfiable.

Kernighan and Lin describe a very successful technique for the graph partitioning problem (44) and a variant of it for the traveling salesman problem (49). Figure 23 shows their algorithm for graph partitioning. Graph partitioning takes as input a graph with $2n$ nodes and partitions the node set into two subsets of size n such that the total weight of the edges connecting nodes in separate sets is minimized. A feasible solution is any equal-size partition. An initial partition is selected by assigning nodes to subsets randomly. The neighborhood structure is defined by the action of selecting one node in each set and swapping them. Move selection proceeds by looking a total of

n moves ahead. It first finds the swap that produces the neighbor of minimum cost (even if this neighbor is worse). It then removes the chosen nodes from consideration and finds another best swap (that is, a neighbor's neighbor). It proceeds until all elements have been swapped (which produces the original partition again), a total of n swaps. Finally, it selects the initial subsequence of these moves that produces the most net improvement. Thus the new solution may be anywhere from 1 to n moves away from the current solution. Note that the sequence selected may include worsening moves. Note, too, that by removing the selected pair at each step, the algorithm effectively forbids moves later on—not just swapping that pair again, but any swap involving either node. Finally, note that the maximum search depth, n , depends on the size of the input. Search stops when no initial subsequence yields a net decrease in cost. The search can be repeated several times from different initial solutions. The solutions found by this algorithm are significantly better than those produced by a straight hill-climbing algorithm (40).

Simulated annealing is a stochastic approach to local search based on an analogy with the annealing of metal (40, 41). The value of this analogy to optimization has been questioned by many researchers, but the terminology of the analogy remains in use. Figure 24 shows a generic simulated annealing algorithm where the neighborhood structure and method of finding an initial solution are unspecified, though the latter usually needs to have a random element since the overall strategy usually involves several searches from different starts. In simulated annealing, a *temperature* parameter is used to influence the move selection process. Improving moves are always accepted, and neutral and worsening moves are stochastically accepted following an exponential distribution based on both the difference in cost between the neighbor and the current solution, and the temperature. This is equivalent to adding noise to the cost function values for non-improving moves. A *run* consists of some number, L , of attempted moves, whether accepted or not. At the end of each run, the temperature is reduced by the *cooling ratio* r , $0 < r < 1$. Search stops when the ratio of accepted moves to run length remains below a threshold for some number of runs with no new best solutions found; the search is said to be *frozen* at this point.


```

procedure GraphPartition
Input: A graph  $G = (V, E)$  where  $|V| = 2n$  and a cost function  $c$  defined over  $E$ 
Output: The best partition  $(A, B)$  of  $V$  found, where  $|A| = |B| = n$ 
begin
  partition  $V$  randomly into subsets  $A$  and  $B$  of size  $n$ 
  loop
     $D := \{ | a \rightarrow D_a \mid (a) a \in A \wedge D_a = \sum_{y \in B} c(a, y) - \sum_{x \in A} c(a, x) \mid \}$ 
     $\cup \{ | b \rightarrow D_b \mid (b) b \in B \wedge D_b = \sum_{x \in A} c(b, x) - \sum_{y \in B} c(b, y) \mid \}$ 
    for  $i := 1$  to  $n$  loop
       $\langle a_i, b_i \rangle := some(a, b) D(a) + D(b) - 2c(a, b) = max(\{ D(a) + D(b) - 2c(a, b) \mid$ 
         $(a, b) a \in A \setminus \{a_j \mid (j) 1 \leq j < i\} \wedge b \in B \setminus \{b_j \mid (j) 1 \leq j < i\} \})$ 
       $g_i := D(a_i) + D(b_i) - 2c(a_i, b_i)$ 
       $D := \{ | a \rightarrow D(a) + 2c(a, a_i) - 2c(a, b_i) \mid (a) a \in A \setminus \{a_j \mid (j) 1 \leq j \leq i\} \mid \}$ 
       $\cup \{ | b \rightarrow D(b) + 2c(b, b_i) - 2c(b, a_i) \mid (b) b \in B \setminus \{b_j \mid (j) 1 \leq j \leq i\} \mid \}$ 
    end loop
     $k := some(k) \sum_{i=1}^k g_i = max(\{ \sum_{i=1}^l g_i \mid (l) 1 \leq l \leq n \})$ 
    exit when  $\sum_{i=1}^k g_i = 0$ 
     $X := \{a_i \mid (i) 1 \leq i \leq k\}$ 
     $Y := \{b_i \mid (i) 1 \leq i \leq k\}$ 
     $A := (A \setminus X) \cup Y$ 
     $B := (B \setminus Y) \cup X$ 
  end loop
  return  $A$  and  $B$ 
end procedure

```

Figure 23. The Kernighan-Lin Algorithm for Graph Partitioning

```

procedure Anneal
Input: A problem instance  $P$ , run length  $L$ , and cooling ratio  $r$ 
Output: The best feasible solution to  $P$  found
begin
  Get an initial feasible solution  $S$  of  $P$ 
   $c := cost(S)$ 
   $S^* := S, c^* := c$ 
  Get an initial temperature  $T > 0$ 
  While not frozen loop
    for  $I$  in 1 ..  $L$  loop
      Pick a random neighbor  $S'$  of  $S$ 
       $c' := cost(S')$ 
       $\Delta := c' - c$ 
      if  $\Delta \leq 0$  then
         $S := S', c := c'$ 
        If  $c' < c^*$  then
           $S^* := S', c^* := c'$ 
        end if
      else
        Choose a random number  $t$  in  $[0, 1]$ 
        if  $t \leq e^{-\Delta/T}$ 
           $S := S', c := c'$ 
        end if
      end if
    end loop
     $T := rT$ 
  end loop
  return  $S^*$ 
end procedure

```

Figure 24. Generic Simulated Annealing Algorithm

Empirical studies show that simulated annealing clearly dominates simple hill-climbing on the graph partitioning problem and compares favorably with the Kernighan-Lin algorithm, even when run times are taken into account (40). For number partitioning, simulated annealing is not competitive with traditional approaches and in fact does no better than simple hill-climbing from random starts, a result blamed on the exceedingly “mountainous” shape of the solution space over the obvious neighborhood structures (41).

Since the 1980’s, Fred Glover and others have introduced and elaborated new approaches to local search that aim to combine elements of artificial intelligence with traditional search techniques to make search more adaptive and effective. These new approaches are collectively known as *tabu*

search (23, 25, 24). Tabu search is characterized by the use of "memory structures" as part of the search state to guide search in various "strategic" ways. The classic example of such a memory structure is the *tabu list*. In the simplest case, a tabu list is a list of moves that are forbidden from being chosen. The basic selection rule is always to make the best move that is not tabu, even if it is to a worse solution. Tabu lists can be used to control the search in many ways, but the most fundamental is for cycle prevention. Once a local optimum is found, only neutral or worsening moves will be available. If a worsening move is made, then at the next step a move that reverses it would be an improving move that should not be taken. Therefore the move back to the solution just visited is declared tabu. This status is maintained for some number of moves before being released, in an attempt to force the search far enough away from the old solution that relaxing the tabu status is safe. The tabu list acts as a FIFO queue, adding newly tabu moves at each search step and dropping the oldest. The length of the list can be static or dynamic, fixed or dependent on the problem instance. Empirical studies are usually required to find appropriate lengths. A further refinement of the tabu mechanism is the use of *aspiration criteria*, which provide means for the tabu status of a move to be overridden, such as when it can be proved that the solution reached is new. The most common of these is to override the tabu status of a move if the solution generated is the best solution seen so far in the search. Stopping criteria for tabu search are usually based on resource limits since improving, neutral and worsening moves are all made freely. Figure 25 shows a generic tabu search algorithm for an unspecified problem.

A tabu list is an example of a short-term, recency-based memory structure. Intermediate and long-term memory structures can also be defined, and can record other kinds of information, such as frequency of solution attributes. This information might be used at times when the search is not progressing, for example to diversify the search by penalizing solutions with high-frequency attributes, or during restarts to influence how future initial solutions are generated. Use of such techniques is independent of the use of short-term memory, but they are still referred to as aspects of tabu search. Some work has been done in integrating memory-based strategies into other search

```

procedure Tabu
Input: A problem instance  $P$ , and possibly tabu parameters
Output: The best feasible solution  $S$  found
begin
  select an initial solution  $S$ 
   $S^* := S$ 
  initialize tabu lists and aspiration criteria
  while stopping criteria not met loop
    create a candidate list of moves
    choose best admissible candidate move, considering tabu status
      and aspiration criteria
    update current solution  $S$ , tabu lists and aspiration criteria
    if  $cost(S) < cost(S^*)$  then  $S^* := S$ 
  end loop
  return  $S^*$ 
end procedure

```

Figure 25. Generic Tabu Search Algorithm

paradigms, such as global search (20) and simulated annealing and genetic algorithms (28). The large volume of published work on tabu search shows it to be a robust and powerful approach.

Genetic algorithms (10, 31, 46, 47) are another stochastic approach and the only one described here that uses a neighborhood relation that is not binary. Like simulated annealing, genetic algorithms are motivated by an analogy, in this case to biological systems, and the terminology was chosen accordingly. Figure 26 shows a simple, generic genetic algorithm. The overall strategy is to maintain a population of solutions through several generations, where each solution competes for the opportunity to breed and thus contribute to future generations. Breeding is accomplished by selecting two solutions and *crossing* them to produce offspring. Solutions chosen for breeding are selected randomly but not uniformly: the chances of a solution being selected are a function of its cost. Thus good solutions are selected preferentially and tend to be bred with other good solutions, but bad solutions are not ignored completely and will occasionally breed. A *mutation rate* set relatively low also perturbs solutions stochastically according to a traditional binary neighborhood relation to maintain a diversified population. Search stops after some number of generations or when the system fails to improve, at which point the best solution is returned. As always, this process can be done several times from different initial populations. The chances of a search state

```

procedure SGA
Input: A problem instance  $P$ , MAX-GENERATIONS
Output: The best feasible solution  $S$  of  $P$  found
begin
  Randomly generate an initial population of feasible solutions
  for  $i$  in 1 .. MAX-GENERATIONS loop
    Generate "new" solutions by breeding, mutating and/or copying
      existing ones stochastically based on their costs
    Replace old population with new one
  end loop
  return best solution found
end procedure

```

Figure 26. A Simple, Generic Genetic Algorithm

recurring are very low, as this amounts to an entire population being identical to that of an earlier step and the seed of the random number generator recurring simultaneously. As search proceeds, however, the population tends to converge on a small number of good solutions and remain fairly stable after that.

Crossover requires that a solution be represented by some kind of fixed-length linear structure called a *chromosome*. Two solutions are crossed by randomly selecting a position along the chromosome and forming a new chromosome (or two, if desired) by combining parts from the two split chromosomes. This implicitly defines a ternary (or quaternary) neighborhood relation on solutions. Genetic algorithms are typically used for unconstrained optimization because it makes defining a suitable crossover operator easier. In (47), however, the traveling salesman problem was solved using a genetic algorithm with a specialized crossover operator that maintained tours (if sequences represent tours, a standard crossover is likely to duplicate some cities and miss others).

Genetic algorithms can be shown to search optimally in a certain sense and often produce satisfactory solutions to difficult, ill-structured problems that defy other solution methods. They are rarely competitive with specialized approaches, however, and can fail miserably (as all search methods can), in particular if the solution representation and crossover operator are poorly fitted to the problem or to the special nature of this approach.

3.5 Problem Relaxation

In some of the above examples there are cases where a problem is modified to make it more amenable to solution. Such problem transformations are an important part of analysis and algorithm design, but the need for them seems especially prevalent when dealing with local search. Relaxation is particularly common.

Relaxation transforms a problem by redefining the space of feasible solutions, in particular by enlarging it. The overall strategy of an algorithm, for example, may involve alternating between different regions of a larger space. Relaxation is often accompanied by a (re)formulation of the cost function. For example, a constraint satisfaction problem can be converted to an optimization problem by relaxing some or all of the constraints and defining a cost function that measures the degree to which a solution violates the constraints that were relaxed. If the optimal solutions to this relaxed problem do not violate the constraints at all, that is they have zero cost, then the original problem is solved. If optimal solutions have non-zero cost, then the original problem has no solution. This use of relaxation makes a much broader class of problems amenable to the application of local search. Examples include local search's own subproblem of finding an initial feasible solution.

Modifying the solution space affects the neighborhood relation, sometimes in ways we can exploit. For some constraints it is relatively easy to find a neighborhood structure that holds them invariant while for others it is more difficult. Relaxing hard constraints may make finding a neighborhood structure easier and/or make the steps of the search more efficient. Sometimes a relaxed space is smoother, with fewer local optima, and hence local search over it is better behaved. It is not uncommon to take relaxation to its logical extreme by formulating problems as unconstrained optimizations. This tends to make a wealth of neighborhood structures available, and also greatly simplifies the problem of finding an initial feasible solution.

When an optimization problem is relaxed, the solution found may be infeasible. Sometimes the analyst will decide that infeasible solutions are acceptable (for example, when dealing with large quantities of discrete objects, the integrality constraint can be relaxed with little impact on the solution). In general, however, something extra must be done in order to return only feasible solutions. If the cost function is modified, the weight assigned to constraint violations must be carefully balanced against the original objectives of the search. Too low a penalty will yield an infeasible optimal solution, while too high a penalty makes the neighborhood more mountainous, with the feasible solutions standing out sharply against a background of infeasible ones, making the good feasible solutions hard to distinguish from the bad and producing large numbers of local optima. Lagrangian relaxation is a technique used in mathematical programming for finding the right weights for constraint violations (2). Another approach is simply to let the search range freely over the enlarged space while keeping track of the best feasible solution seen. For particular problems there may also be techniques available for repairing infeasible solutions that seem promising. Finally, solutions to the relaxed problem may not be used directly at all, but only to provide bounds on optimal solutions to the original problem.

3.6 Conclusion

This chapter has described the characteristic features of local search and surveyed several important classes of algorithms. Chapters IV-VII will formalize a theory of local search suitable for implementation in a system like SPECWARE. This theory is limited to binary neighborhoods and hence excludes genetic algorithms. It also neglects relaxation, or rather assumes the user will know how to do it and recognize when it is appropriate. The theory does encompass the remaining techniques mentioned: hill climbing, simulated annealing, the Kernighan-Lin heuristic, and tabu search.

IV. Algebraic Algorithm Design

The KIDS system pioneered an approach to automating algorithm design based on algorithm theories and design tactics, as described in Chapter I. We seek to continue formalizing and generalizing the concepts and techniques underlying design tactics into an algebraic formalism suitable for implementation in a system such as SPECWARE. This chapter describes the means developed for representing problems and solutions and for storing and manipulating knowledge about algorithmic structures in a uniform and consistent way that is easy to extend. This development is a major research contribution.

4.1 Specifying Problems

4.1.1 A Canonical Form for Problem Specification. KIDS provides syntactic forms in its Regroup language for four kinds of problem specification. The most fundamental of these is captured in the spec *Problem*, which characterizes a problem by its domain or input sort, D , range or output sort, R , input condition, I , which describes which elements of D are valid inputs, and output condition, O , which describes which elements of R are valid outputs with respect to a given input.

```
spec Problem is
  sorts D, R
  op I : D → Boolean
  op O : D, R → Boolean
end-spec
```

This spec provides a canonical form for all problem specifications. A particular problem is specified by giving an interpretation from the *Problem* spec to some domain theory spec. Figure 27 shows an interpretation

$$\textit{Problem} \longrightarrow \textit{BoolSat} \xleftarrow{d} \textit{PropLogic}$$

from *Problem* to the *PropLogic* spec, defined in Section 2.1.4 of Chapter II, that defines the boolean satisfiability problem. The mediator spec, *BoolSat*, imports *PropLogic* and extends it with


```

spec BoolSat is
  import PropLogic

  op I : Formula → Boolean
  definition of I is
    axiom  $\forall (f : \text{Formula}) I(f)$ 
  end-definition

  op DefinedOver : Formula, Assignment → Boolean
  definition of DefinedOver is
    axiom  $\forall (f : \text{Formula}, a : \text{Assignment}) (\text{DefinedOver}(f, a) \Leftrightarrow$ 
       $\forall (p : P) (\exists (c : \text{Clause}) (f\text{-in}(c, f) \wedge (c\text{-in}(\text{pos}(p), c) \vee c\text{-in}(\text{neg}(p), c)))$ 
       $\Leftrightarrow \text{defined-at?}(a, p)))$ 
  end-definition

  op SatisfiesClause : Clause, Assignment → Boolean
  definition of SatisfiesClause is
    axiom  $\forall (c : \text{Clause}, a : \text{Assignment}) (\text{SatisfiesClause}(c, a) \Leftrightarrow$ 
       $\exists (t : T, a\text{-at-}t : (\text{Assignment}, P) \mid \text{defined-at?})$ 
       $(c\text{-in}(t, c) \wedge \langle a, \text{var-of}(t) \rangle = (\text{relax defined-at?})(a\text{-at-}t)$ 
       $\wedge \text{val-of}(t) = \text{map-apply}(a\text{-at-}t)))$ 
  end-definition

  op Satisfies : Formula, Assignment → Boolean
  definition of Satisfies is
    axiom  $\forall (f : \text{Formula}, a : \text{Assignment})$ 
       $(\text{Satisfies}(f, a) \Leftrightarrow \forall (c : \text{Clause}) f\text{-in}(c, f) \Rightarrow \text{SatisfiesClause}(c, a))$ 
  end-definition

  op OPL : Formula, Assignment → Boolean
  definition of OPL is
    axiom  $\forall (f : \text{Formula}, a : \text{Assignment})$ 
       $(\text{OPL}(f, a) \Leftrightarrow \text{DefinedOver}(f, a) \wedge \text{Satisfies}(f, a))$ 
  end-definition
end-spec

interpretation BoolSat : Problem  $\Rightarrow$  PropLogic is
  mediator BoolSat
  domain-to-mediator  $\{D \rightarrow \text{Formula}, R \rightarrow \text{Assignment}, O \rightarrow \text{OPL}\}$ 
  codomain-to-mediator import-morphism

```

Figure 27. Boolean Satisfaction as a Problem in Propositional Logic

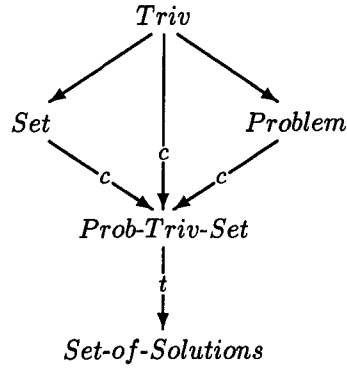
definitions describing valid formulas (all formulas are valid) and valid assignments (an assignment solves a formula if it is defined over the variables of the formula and it satisfies each clause). Note that there can be several interpretations from *Problem* into a domain theory spec, defining distinct problems in a single domain. Specifying problems in this form indicates unambiguously the problem to be solved.

4.1.2 A Canonical Form for Finding All Solutions to a Problem. The second form of problem specification recognized by KIDS is the case where a problem has multiple solutions and we want to compute all of them and return a set. Figure 28 illustrates how to form a spec *Set-of-Solutions* as the colimit of a simple diagram. This spec adds to *Problem* the sort and operations that define a set of solutions. We also define an interpretation *Find-All-Solutions*,

$$Problem \longrightarrow All-Solutions \longleftarrow d \longleftarrow Set-of-Solutions$$

to express explicitly what finding all solutions means, using the canonical form for problems defined above. Figure 29 shows the details of this interpretation. The mediator spec and the source morphism define the domain and input condition to be those of the embedded problem, while the range is a set of solutions and the output condition says that such a set is valid if and only if it contains exactly the solutions to the given input.

The interpretation *Find-All-Solutions* provides a canonical form for the class of problems of the find-all-solutions type: all the user has to specify is the particular problem embedded in it. Figure 30 illustrates how to build a specification of the problem of finding all satisfying assignments of a formula. On the lower left is *Find-All-Solutions*. Above the spec *Set-of-Solutions* is shown its internal structure as the translation of a colimit. We will use this structure to refine *Set-of-Solutions* via diagram refinement, which was defined in Chapter II. The diagram is refined by attaching to each node and arc an interpretation and an interpretation morphism, respectively. The nodes labeled *Set* and *Triv* have identity interpretations for their respective specifications, and the interpretation morphism between them is three copies of the $Triv \rightarrow Set$



spec *Prob-Triv-Set* **is**
 colimit of diagram
 nodes *Triv, Set, Problem*
 arcs *Triv* $\rightarrow$ *Set* : $\{\}$, *Triv* $\rightarrow$ *Problem* : $\{E \rightarrow R\}$
 end-diagram

spec *Set-of-Solutions* **is**
 translate *Prob-Triv-Set* by $\{E \rightarrow R, Set \rightarrow RSet\}$

Figure 28. Constructing a Spec for a Set of Solutions

spec *All-Solutions* **is**
 import *Set-of-Solutions*
 op *OAll* : *D, RSet* $\rightarrow$ *Boolean*
 definition of *OAll* **is**
 axiom $\forall(x : D, ZSet : RSet)$
 $OAll(x, ZSet) \Leftrightarrow \forall(z : R) (z \in ZSet \Leftrightarrow O(x, z))$
 end-definition
end-spec

interpretation *Find-All-Solutions* :
 Problem $\Rightarrow$ *Set-of-Solutions* **is**
 mediator *All-Solutions*
 domain-to-mediator $\{R \rightarrow RSet, O \rightarrow OAll\}$
 codomain-to-mediator **import-morphism**

Figure 29. Problem Specification for Finding All Solutions

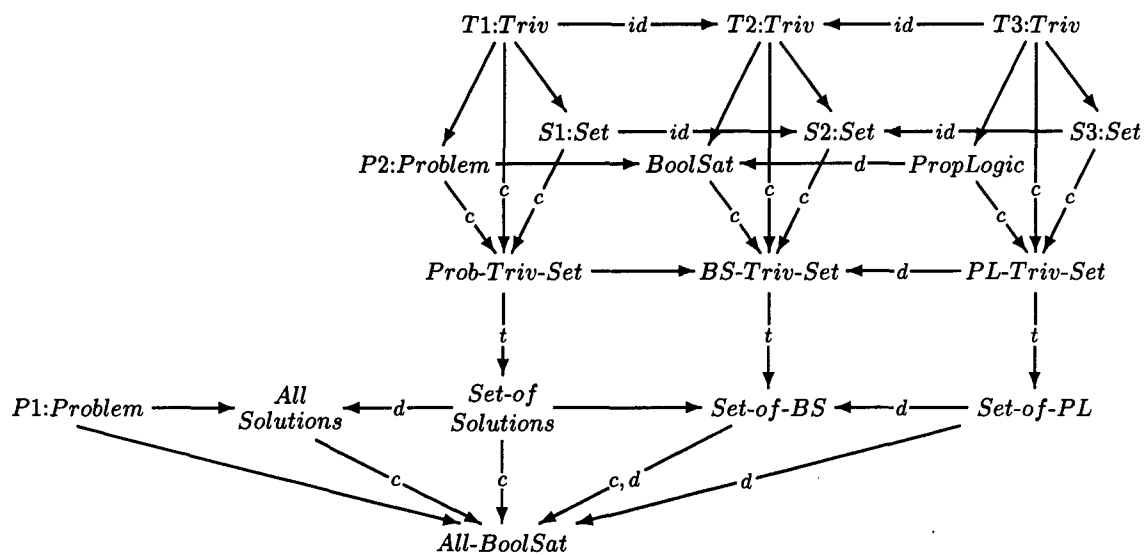


Figure 30. Constructing a Specification for All-BoolSat Problem

morphism. The specification for boolean satisfiability is attached to the *Problem* node. The interpretation morphism from the *Triv* identity interpretation maps E to *Assignment* for both the mediator and target specifications. This diagram of refinements has a colimit in the form of an interpretation among the respective colimit specs, shown as $Prob-Triv-Set \Rightarrow PL-Triv-Set$. A translation comparable to $Prob-Triv-Set \xrightarrow{t} Set-of-Solutions$ is then applied to each of the other colimit specs, to rename the *Set* sort in each spec appropriately. The result is the interpretation $Set-of-Solutions \Rightarrow Set-of-PL$. Finally, this interpretation is composed with *Find-All-Solutions* as explained in Chapter II to yield a single interpretation $Problem \Rightarrow Set-of-PL$ that specifies the desired problem, with spec *All-BoolSat* as the mediator. We will see below that structuring the problem specification in terms of an existing problem class in this fashion is an important facet of algorithm design.

4.1.3 Canonical Forms for Global Optimization Problems. Another important problem class, again recognized by special syntax in KIDS, is global optimization. A global optimization problem consists of a space of *feasible solutions* and a *cost* or *objective function* defined over this space. The goal is to find a feasible solution whose cost is at least as good as that of every other

```

spec CostOrder is
  translate TotalOrder by  $\{E \rightarrow \mathcal{R}\}$ 

spec WFSS is
  import Problem, CostOrder
  op Cost :  $D, R \rightarrow \mathcal{R}$ 
end-spec

spec GlobalOptimum is
  import WFSS

  op Optimal :  $D, R \rightarrow \text{Boolean}$ 
  definition of Optimal is
    axiom  $\forall(x : D, y : R) (Optimal(x, y) \Leftrightarrow$ 
       $\forall(y' : R) (O(x, y') \Rightarrow Cost(x, y) \leq Cost(x, y')))$ 
  end-definition

  op GO :  $D, R \rightarrow \text{Boolean}$ 
  definition of GO is
    axiom  $\forall(x : D, y : R) (GO(x, y) \Leftrightarrow O(x, y) \wedge Optimal(x, y))$ 
  end-definition
end-spec

interpretation GlobalOptimization : Problem  $\Rightarrow$  WFSS is
  mediator GlobalOptimum
  domain-to-mediator  $\{O \rightarrow GO\}$ 
  codomain-to-mediator import-morphism

```

Figure 31. Canonical Form for Global Optimization Problems

feasible solution. The spec *WFSS*, shown in Figure 31, is a domain theory for this class. It imports *Problem* to define the space of feasible solutions and *CostOrder* to provide a codomain for the cost function, introduced as the operation *Cost*. A totally ordered sort is needed so that costs can be compared. Without loss of generality, we use the symbol $\leq$ to represent the desired ordering.

The global optimization problem class is specified by the interpretation *GlobalOptimization*,

$$Problem \longrightarrow GlobalOptimum \longleftarrow_d WFSS$$

also shown in Figure 31. A particular optimization problem would be specified by giving an interpretation from *WFSS* to a problem domain and then composing it with *GlobalOptimization*. This construction is more straightforward than the find-all-solutions example; diagram refinement is unlikely to be needed.

The final problem class directly recognized by KIDS is finding all global optima. We can form a specification for this problem class by combining *Find-All-Solutions* with *GlobalOptimization*, using diagram refinement to add sets of R as we did for *All-Boolsat* in Figure 30. In this case, the range sort R is a set of globally optimal solutions to the embedded feasibility problem. An instance of this problem class would be specified by providing an interpretation from *WFSS*, carrying out another diagram refinement and then composing the three resulting interpretations (pairwise, in any order).

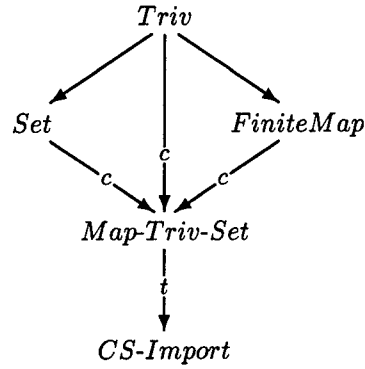
4.1.4 A Canonical Form for Constraint Satisfaction Problems. The representation of problem classes by interpretations from *Problem* can be carried out for other problem classes, such as constraint satisfaction or integer programs. Figures 32 and 33 show a specification *Constraint-Satisfaction*,

$$Problem \longrightarrow CSP \longleftarrow^d CS$$

for a class of constraint satisfaction problems given in (71). This class posits the existence of two sorts, Var and Val , representing variables and values, respectively. The output sort is a map from variables to values. The input sort and input condition are uninterpreted symbols D and I , respectively, and the operation *Variables* is intended to extract from an input the set of variables that need values. All variables are assumed to have the same set of legal values, described by the predicate *LegalVal*. Finally, the remaining constraints to be satisfied are described by the *OCS* operation. The output condition for constraint satisfaction, *OCSP*, is defined in the mediator as a map that is defined over the variables of the input, assigns each variable a legal value, and satisfies the *OCS* constraints.

Boolean satisfiability can be expressed as a member of this class. Figure 34 shows an interpretation *CS-BoolSat*,

$$CS \longrightarrow CS-as-PL2 \longleftarrow^d PropLogic2$$



```

spec Map-Triv-Set is
  colimit of diagram
    nodes Triv, Set, FiniteMap
    arcs  Triv → Set : {},
          Triv → FiniteMap : {E → Dom}
end-diagram

spec CS-Import is
  translate Map-Triv-Set by
    {Dom → Var, Cod → Val, Map → Assignment, Set → VarSet}

spec CS is
  import CS-Import

  sort D

  op I : D → Boolean
  op Variables : D → VarSet
  op LegalVal : D, Var, Assignment → Boolean
  op OCS : D, Assignment → Boolean
end-spec

```

Figure 32. Constraint Satisfaction Domain Theory

```

spec CSP is
  import CS

  op OCSP : D, Assignment  $\rightarrow$  Boolean
  definition of OCSP is
    axiom  $\forall(x : D, a : \text{Assignment}) (OCSP(x, a) \Leftrightarrow$ 
       $\forall(v : \text{Var}) (in(v, \text{Variables}(x)) \Leftrightarrow \text{defined-at?}(a, v))$ 
       $\wedge \forall(v : \text{Var}) (\text{defined-at?}(a, v) \Rightarrow \text{LegalVal}(x, v, a))$ 
       $\wedge OCS(x, a))$ 
    end-definition
  end-spec

  interpretation ConstraintSatisfaction : Problem  $\Rightarrow$  CS is
    mediator CS
    domain-to-mediator  $\{R \rightarrow \text{Assignment}, O \rightarrow OCSP\}$ 
    codomain-to-mediator import-morphism

```

Figure 33. Specification of Constraint Satisfaction Problem

that shows how the elements of constraint satisfaction can be used to express boolean satisfiability. Note that a slightly different domain theory, *PropLogic2*, is used. This was needed to add sets of variables to the domain theory. A complete specification for boolean satisfiability is obtained as with other canonical forms by composing *ConstraintSatisfaction* with *CS-BoolSat*.

4.1.5 The Value of Canonical Forms. The algebraic method for defining problem classes is more general than the specialized syntax required by KIDS and allows new classes to be added to the knowledge base easily, as we just illustrated with constraint satisfaction. Knowledge that a particular problem belongs to some problem class allows us to use additional knowledge about the class and how its structure can be exploited to generate efficient algorithms for solving it. Before describing how this is done, we first need a generic representation for algorithms or programs.


```

spec PropLogic2 is
  translate colimit of diagram
    nodes Triv, Set, PropLogic
    arcs Triv → Set : {}, Triv → PropLogic : {E → P}
  end-diagram by
    {Set → PSet, NE-Set → NE-PSet, delete → p-delete, empty-set → empty-pset,
     nonempty-set? → nonempty-pset?, in → p-in, insert → p-insert,
     singleton → p-singleton, union → p-union}

spec CS-as-PL2 is
  import PropLogic2

  op I : Formula → Boolean
  definition of I is axiom  $\forall(f : \text{Formula}) I(f)$  end-definition

  op Variables : Formula → PSet
  definition of Variables is
    axiom  $\forall(f : \text{Formula}, p : P)$ 
       $(p\text{-in}(p, \text{Variables}(f))$ 
         $\Leftrightarrow \exists(c : \text{Clause}) f\text{-in}(c, f) \wedge (c\text{-in}(\text{pos}(p), c) \vee c\text{-in}(\text{neg}(p), c)))$ 
    end-definition

  op LegalVal : Formula, P, Assignment → Boolean
  definition of LegalVal is
    axiom  $\forall(f : \text{Formula}, p : P, a : \text{Assignment}) \text{LegalVal}(f, p, a)$ 
  end-definition

  op SatisfiesClause : Clause, Assignment → Boolean
  definition of SatisfiesClause is
    axiom  $\forall(c : \text{Clause}, a : \text{Assignment}) (\text{SatisfiesClause}(c, a) \Leftrightarrow$ 
       $\exists(t : \text{Term}, a\text{-at-}t : (\text{Assignment}, \text{Term}) \mid \text{defined-at-}t?)$ 
       $(c\text{-in}(t, c) \wedge \langle a, \text{var-of}(t) \rangle = (\text{relax defined-at-}t?)(a\text{-at-}t)$ 
       $\wedge \text{val-of}(t) = \text{map-apply}(a\text{-at-}t)))$ 
    end-definition

  op Satisfies : Formula, Assignment → Boolean
  definition of Satisfies is
    axiom  $\forall(f : \text{Formula}, a : \text{Assignment})$ 
       $(\text{Satisfies}(f, a) \Leftrightarrow \forall(c : \text{Clause}) f\text{-in}(c, f) \Rightarrow \text{SatisfiesClause}(c, a))$ 
    end-definition
  end-spec

interpretation CS-BoolSat : CS  $\Rightarrow$  PropLogic2 is
  mediator CS-as-PL2
  domain-to-mediator
    {D → Formula, Var → P, VarSet → PSet, Val → Boolean, OCS → Satisfies,
     NE-Set → NE-PSet, delete → p-delete, empty-set → empty-pset,
     nonempty-set? → nonempty-pset?, in → p-in, insert → p-insert,
     singleton → p-singleton, union → p-union}
  codomain-to-mediator import-morphism

```

Figure 34. Boolean Satisfiability Expressed as Constraint Satisfaction

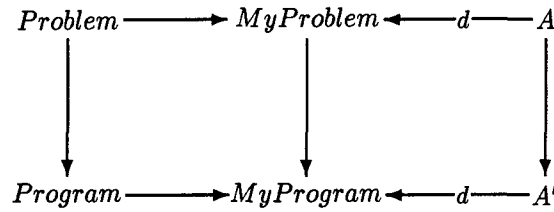


Figure 35. Canonical Form for a Solution to a Problem

4.2 Specifying Solutions

A canonical form for the (computational) solution to a problem is provided by

```

spec Program is
  sorts D, R
  op I : D → Boolean
  op O : D, R → Boolean
  op F : D → R
  axiom correctness is  $\forall(x : D) (I(x) \Rightarrow O(x, F(x)))$ 
end-spec

```

This spec extends *Problem* by adding an operation *F* that computes outputs from inputs, and an axiom formalizing what correctness means: valid inputs are mapped to valid outputs. The behavior of *F* on invalid inputs is unspecified. As for problems, a particular program is specified by giving an interpretation from *Program*. A solution is related to the problem it solves by an interpretation morphism as shown in Figure 35. An interpretation morphism is a set of three morphisms from one interpretation to another such that the resulting diagram commutes. It shows how one interpretation is embedded in or refined by another. In this case, the fact that the left square in particular commutes (that is, that *D, R, I* and *O* are mapped to the same elements by the composition of *Problem* → *MyProblem* with *MyProblem* → *MyProgram* as by the composition of *Problem* → *Program* with *Program* → *MyProgram*) guarantees that the solution specified by the bottom interpretation solves the problem specified by the top one. The domain theory *A* is shown extended to *A'* in the solution to represent the possible addition of new sorts and operations to support computation.

It should be noted that Figure 35 gives only the form of a solution to a problem. There is no unique solution, nor a minimal, most general or best solution. There are even “solutions” of the proper form that are useless from a computational perspective. In the extreme, A can be extended with an element F and an axiom declaring the correctness of F . What is intended, of course, is that the definition of F in *MyProgram* be constructive, either in the technical sense required by SPECWARE for code generation or in a looser sense meaning an algorithm has been described.

4.3 Algorithm Design

The goal of algebraic algorithm design is to construct from a problem specification in canonical form the interpretation morphism shown in Figure 35. Design proceeds in two phases. In the first phase a problem is classified as belonging to a particular problem class possessing a certain structure for which one or more solution methods are known. These methods are represented by interpretation morphisms from the problem class specification to a solution specification. The second phase of design is to instantiate a selected solution method with the particulars of the problem being solved. Each of these phases will be described in more detail.

Figure 36 shows a generic classification step. The classification process “factors” a problem specification (the interpretation forming the top of the square) into a generic part representing a known class of problems and the details of a particular instance of the class (the interpretations forming respectively the left side and bottom of the square). As before in discussing solutions, it may be necessary to extend the domain theory A to support the classification. The domain theory for the problem class is here called an *algorithm theory*, and named AT , to emphasize its role in algorithm design; the mediator $ATProblem$ defines what an AT -problem is. An interpretation morphism relates the original specification $Problem \Rightarrow A$ to the middle interpretation $ATProblem \Rightarrow A'$, insuring that it specifies the same problem. The definitional extension $AT \xrightarrow{d} ATProblem$ provides a simpler target spec for the problem class and serves to simplify the mediator similarly, to $MyAT$.

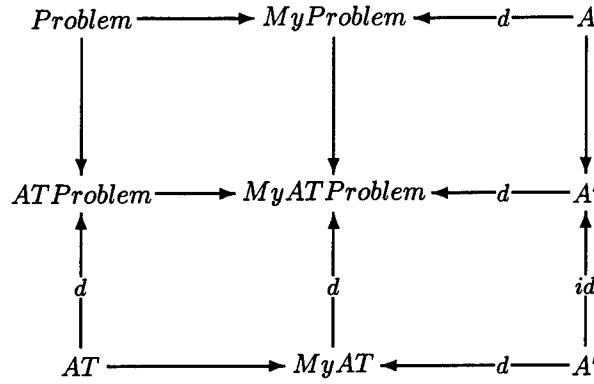


Figure 36. Problem Classification

The bottom interpretation morphism, composed entirely of definitional extensions, insures that the simplified form is consistent with the middle interpretation and hence with the original problem specification.

Figure 37 shows the classification diagram for boolean satisfiability as a constraint satisfaction problem, relating the two specifications for this problem given above. The original specification is shown at the top of the diagram, and the specification for constraint satisfaction problems is on the left. The interpretation mapping constraint satisfaction to the propositional logic domain is on the bottom. New in this diagram are the center spec, *CSP-as-PL2*, and the various morphisms into it, which serve to insure that the factored specification is equivalent to the original one. Details of the additions are shown in Figure 38. Note that *CSP-as-PL2* contains two definitions for *O*, one copied from *BoolSat* and one from *CSP*. This makes the morphisms from these specs easy to verify, but imposes an obligation to prove that the two definitions are equivalent. In this case the proof is straightforward, but in general proving two functions are equivalent is undecidable. Once this equivalence is established, it is easy to prove that the four small squares in the corners of the diagram all commute, and thus that the diagram truly consists of interpretation morphisms.

Figure 37 shows the square that is most useful for classifying boolean satisfiability as constraint satisfaction. This bears emphasis because it is not the only square possible. As with

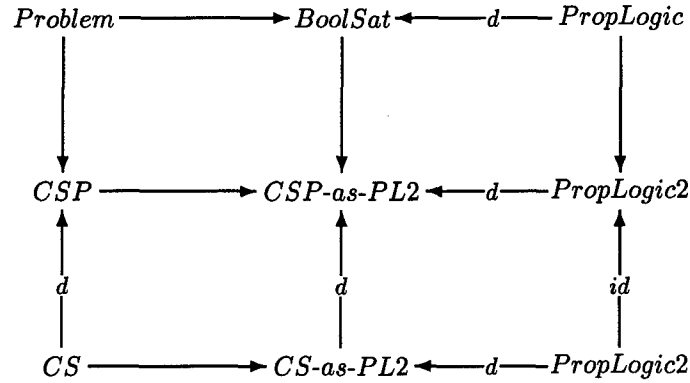


Figure 37. Boolean Satisfiability Classified as Constraint Satisfaction

```

spec CSP-as-PL2 is
  import CS-as-PL2

  op DefinedOver : Formula, Assignment → Boolean
  definition of DefinedOver is
    axiom  $\forall(f : \text{Formula}, a : \text{Assignment})$ 
       $(\text{DefinedOver}(f, a) \Leftrightarrow \forall(p : P) (p\text{-in}(p.\text{Variables}(f)) \Leftrightarrow \text{defined-at?}(a, p)))$ 
  end-definition

  op O : Formula, Assignment → Boolean
  definition of O is
    axiom  $\forall(f : \text{Formula}, a : \text{Assignment})$ 
       $(O(f, a) \Leftrightarrow \text{DefinedOver}(f, a) \wedge \text{Satisfies}(f, a))$ 
  end-definition

  definition of O is
    axiom  $\forall(f : \text{Formula}, a : \text{Assignment}) (O(f, a) \Leftrightarrow$ 
       $\forall(p : P) (p\text{-in}(p, \text{Variables}(f)) \Leftrightarrow \text{defined-at?}(a, p))$ 
       $\wedge \forall(p : P) (\text{defined-at?}(a, p) \Rightarrow \text{LegalVal}(f, p, a)) \wedge \text{Satisfies}(f, a))$ 
  end-definition
end-spec

morphism BoolSat-to-Center : BoolSat → CSP-as-PL2 is {OPL → O}

morphism CSP-to-Center : CSP → CSP-as-PL2 is
  {D → Formula, Var → P, VarSet → PSet, Val → Boolean, OCS → Satisfies,
   OSCP → O, NE-Set → NE-PSet, delete → p-delete, empty-set → empty-pset,
   nonempty-set? → nonempty-pset?, in → p-in, insert → p-insert,
   singleton → p-singleton, union → p-union}

```

Figure 38. Details of Boolean Satisfiability Classification

solutions represented as interpretation morphisms, Figure 36 gives the form of a classification step without capturing its full semantic intent. There is no unique classification diagram, even for a given problem class, and some diagrams of the proper form do not represent useful classifications. Figure 39 shows another square for “classifying” boolean satisfiability as constraint satisfaction. Its construction uses a technique that could be applied to an arbitrary pair of interpretations, even when no meaningful relationship between them exists. The center object is simply the pushout of the top and left mediators and their common source spec. This spec could be duplicated for the remaining specs, but instead simple “clean up” rules were applied to simplify the other three specs by removing unneeded definitions, as the names chosen are meant to suggest (they are not of course valid Slang identifiers). (The $\oplus$ notation subscripted with a spec name indicates a pushout.) As above, the center spec may be inconsistent if the two definitions for O cannot be proved equivalent (or in the general case if any shared sort or operation is defined inconsistently). This proof obligation places constraints on the undefined elements from CS , but whether they are tight enough to provide definitions for these elements is certainly not clear to the casual observer. CS and $PropLogic$ both contain a copy of *FiniteMap*, so the center object contains two copies. Because of *Problem*’s role in the pushout, the two map sorts and R have all been identified as one sort. There is no formal mechanism known, however, for concluding that P is to be identified with Var , or *Boolean* with Val . Without these identifications, and without definitions for *Variables*, *LegalVal* and *OCS*, this valid square is not useful for classification. This method of combination is more akin to a colimit, where objects are combined but do not interact. We seek something less symmetrical, where a stronger effort is made to define T in terms of A , extending A as little as possible.

Still other squares are possible. In Figure 37, *PropLogic* is extended to *PropLogic2* by adding a copy of *Set* to provide sets of sort P . It is possible, however, to implement sets of P using sorts and operations already in *PropLogic*. In particular, since a formula is a set of a set of terms for which the *var-of* operation returns a variable, this is easily done. By adding definitions to the bottom mediator to simulate sets of P using the operations for *Formula*, one can construct a

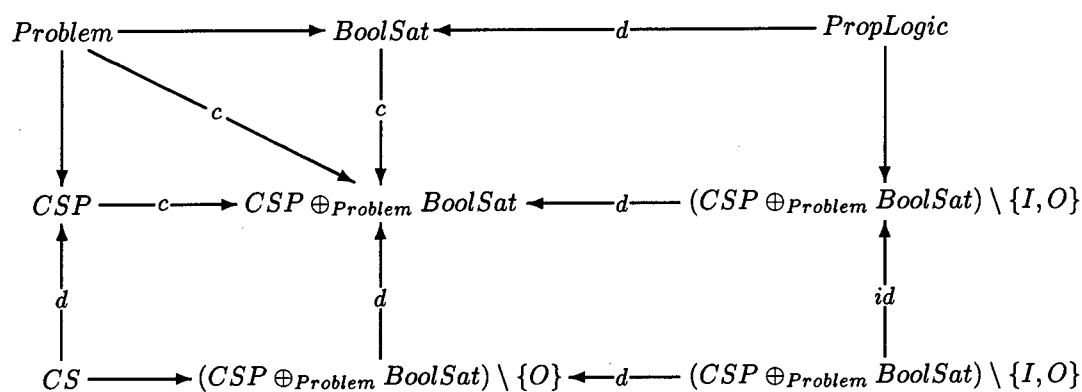


Figure 39. Alternative Classification Diagram for *BoolSat*

square that does not extend *PropLogic* at all. Taking advantage of this circumstance, however, would be a design error. The two sets are used for quite different purposes and might during design be implemented or optimized differently. Identifying sets of *P* with *Formula* early in design at worst prevents an efficient implementation and at best causes more work later to undo it. This example shows that it is possible to make too strong an effort to minimize the extensions to *A*.

Sometimes a problem class can be represented by a morphism from *Problem* rather than an interpretation. Morphisms can be raised to interpretations by adding a second copy of the target spec as the target of the interpretation and connecting it to the first copy, which becomes the mediator, by an identity morphism. In the classification diagram above, the result is that the middle and bottom interpretations are identical, and the bottom interpretation morphism is composed of identity morphisms. In this case, it is convenient to drop all the duplicate specs and use only the top interpretation morphism. We will see examples of this later.

As mentioned in Chapter I, problem classes can be arranged in a refinement hierarchy and multiple classification steps can be made before moving to the next phase of design, with each step progressively calling out more problem structure. Figure 36 illustrates one “rung” of a classification ladder as proposed by Smith (71). We have identified classification as an integral part of algorithm design, not an optional approach to follow or merely a means to reuse existing software.

Classification is a creative process that can be supported but not automated with current technology. For a global optimization problem, for example, the most natural way to specify the problem is to make use of *WFSS* and the interpretation *GlobalOptimization* given above that defines this class; thus the user does the classification. For boolean satisfiability, we saw above that one can specify the problem directly or one can make use of a pre-existing domain theory for constraint satisfaction problems. If the former approach is used, it may still be desirable to recast boolean satisfiability as constraint satisfaction in order to exploit its features. Restructuring an existing problem specification in order to classify it is in general a very hard problem. Chapter V will discuss some techniques for accomplishing the classification step.

The second phase in algorithm design is to assemble a program. KIDS does this by instantiating a program scheme written in the Regroup language, replacing place holders for the sorts and operations of the algorithm theory with the definitions derived in the classification phase. Figure 40 shows the algebraic equivalent of a program scheme, which is an interpretation morphism from a problem class specification to a program specification. The morphism $AT \rightarrow AT'$ represents problem-independent extensions needed by the program scheme to support computation. These extensions can be added to $AT \Rightarrow A'$, derived in phase one, by means of a colimit. That is, $MyAT'$ is formed by taking the pushout of $AT \rightarrow MyAT$ and $AT \rightarrow AT'$. Then A'' is formed from $MyAT'$ by removing some unneeded definitions from it, namely the definitions that $MyAT$ added to A' . The result is two interpretation morphisms as shown in Figure 41 that can be composed in parallel to produce the interpretation morphism called for in Figure 35. Instantiation is a purely syntactic operation that can be fully automated once the user has selected a program scheme.

4.4 Algorithm Refinement

We have been speaking of algorithm design in the context of identifying a problem in a domain theory, A . The domain theory provides background definitions and the problem specification calls

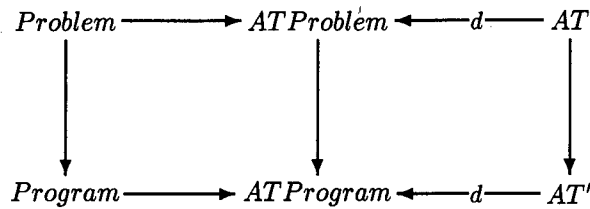


Figure 40. An Algebraic Program Scheme

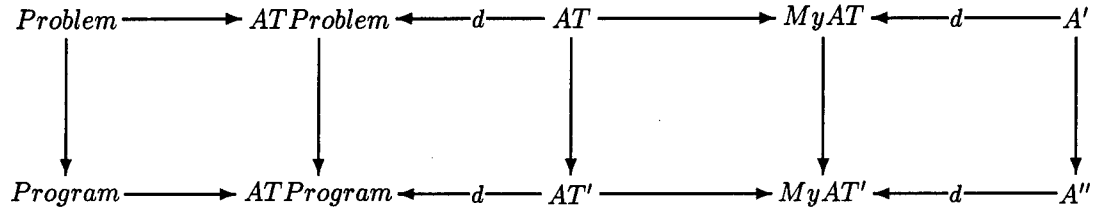


Figure 41. Instantiating a Program Scheme

out a particular problem, which we then solve. This adds a new operation F , a definition for it, and a theorem (the translation of the correctness axiom in *Program*) describing the relevant aspects of its behavior (i.e., its behavior on valid inputs).

Not all specs are intended to be interpreted as domain theories, however. Indeed, once F is introduced the spec is intended as a description of a system to be built. Much of the work on algebraic specification is directed toward specifying systems, not just defining domain theories (17, 30, 43). Certainly if SPECWARE is intended to cover all software activities, it must support both purposes, and its code-generation capabilities are directed to this end.

This leads to the need for a capability to design an algorithm for a particular operation or expression in a spec. A specification for an air traffic control system, for example, might posit an operation for assigning a radar return to an existing track. The specification will describe the essential properties of this operation, but not an algorithm for carrying it out. Another example looks ahead to program schemes for local search, where one expects to see an operation introduced for finding the initial solution. This operation must compute a feasible solution, but the program scheme need not specify how. Indeed it should not, for it is preferable to have the ability to apply

an arbitrary solution technique to this subproblem. Many variants of local search rely on a greedy technique that requires finding for the current solution the neighbor of best cost. This requirement may be part of the definition of a move selection operation and not correspond to any named operation, yet still require a nontrivial algorithm to find the minimum efficiently (solving this subproblem for a linear program by formulating it as an embedded linear program is the basis for Dantzig-Wolfe decomposition (45)). In each case, we need a technique for designing an algorithm for a problem and associating the resulting F with the original operation or unnamed expression. Since this process can be used to refine an existing algorithm, it is called *algorithm refinement*. It is one way in which the principle of *top-down design* can be implemented in an algebraic setting.

Figure 42 shows how algorithm refinement is carried out. Let A be a spec containing some operation or expression for which we wish to design an algorithm. The first step is to specify a problem in the canonical form for algorithm design: extend A as needed to provide named sorts and operations for the domain and range sorts and input and output conditions of the desired operation, and define a morphism from *Problem* to this new mediator spec. This corresponds to the top interpretation in the figure. The second step is to devise a solution to this problem, corresponding to the interpretation morphism to $Program \Rightarrow A'$. These two steps are standard algorithm design as described above. The third step then is to integrate the solution back into the original spec. Create a new spec A'' starting from a copy of *MyProgram*. If F implements an existing operation G in A' , then use the definition of F as the definition of G in A'' and drop the name F . If F implements an unnamed expression in A' , then substitute a call to F for every occurrence of that expression throughout A'' . Define morphisms $MyProgram \rightarrow A''$ and $A' \rightarrow A''$. These both exist because F correctly implements the behavior required of G or of the unnamed expression. Moreover, the triangle formed by these two morphisms and $A' \rightarrow MyProgram$ will commute. The formation of A'' does not correspond to any standard algebraic or categorical operation, but is easily automated.

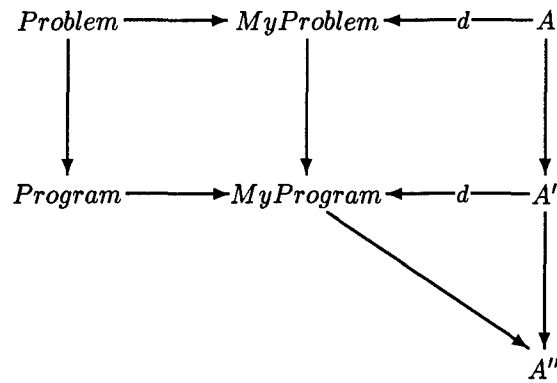


Figure 42. General Form of Algorithm Refinement

4.5 Conclusion

This chapter, though short, lays the formal foundation for the rest of the dissertation and for algorithm design in general. Canonical forms for problem classes and algorithm theories provide a uniform and extensible means for representing knowledge about problems at an abstract, conceptual level. The classification process provides a formal mechanism for applying this knowledge to particular cases, performing what was called in Chapter I the problem setting task of an engineering process for software. The canonical form for solutions and its realizations in the form of program schemes for particular problem classes and algorithm theories represent knowledge about how problems are solved and are used directly to synthesize solutions to classified problems. The various morphisms, interpretations and interpretation morphisms constructed at each step maintain a complete audit trail throughout classification and synthesis to guarantee that the evolving specification is consistent with the original one.

The remaining chapters all build on this one. Chapter V describes techniques for performing the constructions described above. Chapters VI and VII apply the theory to local search, defining algorithm theories describing classes of problems to which local search is applicable and presenting program schemes for solving members of these classes.

V. Completing Interpretation Morphisms

In Chapter IV we saw several situations in which we desired to complete a partially constructed interpretation morphism. During diagram refinement, we may attach an interpretation to one node and desire to use that refinement to help guide the refinement of another node related by morphism to the first. In algebraic algorithm design, the goal is to take a problem specification in the form of an interpretation and complete it to an interpretation morphism that expresses an algorithmic solution to the problem. The method proposed for doing this comprises two steps, classification and instantiation, both of which involve completion of interpretation morphisms. Recalling Figure 36, we see that each classification step in fact consists of *two* completion steps, corresponding to the top and bottom interpretation morphisms, respectively.

In each case we are given one interpretation directly and the source spec of the other, with a morphism between them. There are two slightly different problems, based on the direction of the morphism between the source specs. If the given interpretation is the source of the interpretation morphism, we have the situation illustrated in Figure 43. If the given interpretation is the target of the interpretation morphism, we have the situation illustrated in Figure 44. Problem classification provides an example of each. To complete the top interpretation morphism, we build the middle interpretation and the mediator and target morphisms from the given top interpretation, the source spec of the middle one and a morphism from top source to middle source. To complete the bottom interpretation morphism, we build the bottom interpretation and two morphisms given the newly constructed middle, the source spec of the bottom and a morphism from bottom source spec to middle source. Diagram refinement also includes instances of each problem. Instantiation of a program scheme is an example of the first case.

As we also saw in Chapter IV, such problems do not have unique solutions. In some cases where the source interpretation is given, for example, we seek to duplicate in the mediator and target specs the extension given for the source in a relatively straightforward way, by simply adding

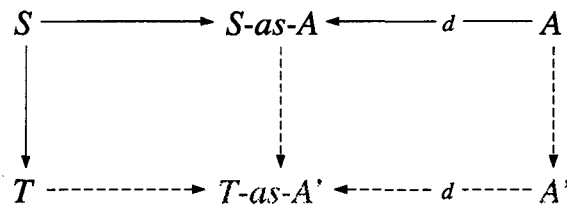


Figure 43. Completing an Interpretation Morphism Given the Source

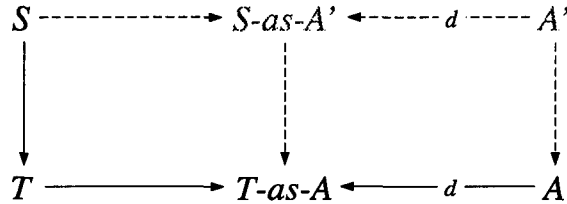


Figure 44. Completing an Interpretation Morphism Given the Target

corresponding pieces to each. Instantiation is an example of this: the extensions added to the algorithm theory are not problem dependent, but serve only to support a particular algorithm. These extensions need to be added to the problem domain to carry out the instantiation. In other cases, a deeper semantic connection is being sought. During classification, for example, we do not seek to add structure but rather to identify structure that (we think) is already there. We do this by extending the mediator with all the added sorts and operations of the new source spec and *defining* them in terms of the existing target spec, extending it relatively little. The extensions made to the target spec may serve only to support these definitions and hence could bear little resemblance to the extensions given for the source spec. As expected, this kind of “semantic extension” is much harder and in particular cases may not be possible at all, whereas syntactic extension is largely mechanical.

The next section of this chapter describes several syntactic approaches to completing interpretation morphisms. Identifying the need for these methods and collecting them together are minor contributions of the research. Following that is a detailed analysis of Smith’s theory of *connections between specifications*, including an intuitive explanation of the underlying principles, an algebraic presentation of how connections are used to complete interpretation morphisms, some notes on

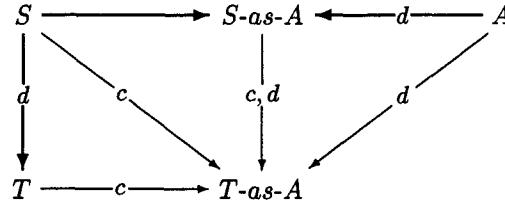


Figure 45. Extending an Interpretation Along a Definitional Extension

practical application of the theory, and an example. Elaboration of the theory of connections and its casting in algebraic terms is a major research contribution.

5.1 Syntactic Methods for Completing Interpretation Morphisms

Completing an interpretation morphism given the source interpretation can be approached in different ways and the choice depends in part on the nature of the morphism that extends the source spec. Extending an interpretation along a definitional extension is an easy task done using a colimit and morphism composition, as shown in Figure 45 (80). $T\text{-as-}A$ is formed as the pushout of $S \rightarrow S\text{-as-}A$ and $S \rightarrow T$. Pushouts preserve definitional extension, so the morphism $S\text{-as-}A \rightarrow T\text{-as-}A$ is also a definitional extension. Definitional extensions compose, so the morphism $A \rightarrow T\text{-as-}A$ is a definitional extension. Thus $T \Rightarrow A$ with $T\text{-as-}A$ as mediator is the unique solution to this special case, and the spec A does not need to be extended at all.

A translation can be viewed as a special case of definitional extension but deserves special treatment. A translation defines an isomorphism from the source spec S to the target spec T : except for the names chosen for the sorts and operations, the two specs are identical. A valid interpretation morphism can be constructed using the same mediator and target specs and identity morphisms. The morphism $T \rightarrow S\text{-as-}A$ is uniquely determined by the requirement that the left square commute: in effect you invert the translation (it has an inverse because it is an isomorphism) and compose it with $S \rightarrow S\text{-as-}A$. Instead of identity morphisms, translations can be applied to the mediator, the target or both. Considering the names in S that are changed by the source

translation, it may be desirable to make corresponding changes to the images of those elements in $S\text{-as-}A$ and in A , if those elements are present in the latter. This could mean copying the name used in T or adopting a new name. An automated tool could use identity morphisms as the default and give the user the option of changing them to other translations. The naming conventions proposed in (71) could also be implemented as the default translations. There is a certain elegance to using translations uniformly in this way, even if they are identities. In diagram refinement, for instance, the result is an attractive parallelism of structure in the final refinement, from source to mediator to target. Figure 30 illustrates the idea, where a copy of *Set* is added uniformly across a refinement and then uniformly renamed to something more suitable.

In the general case where $S \rightarrow T$ is not a translation or a definitional extension, the interpretation morphism can be completed in distinct ways that are not equivalent. *Identity completion* is a syntactic approach to duplicating in the mediator the extensions described by $S \rightarrow T$. It is illustrated in Figure 46. As for definitional extensions, the mediator of the target interpretation is formed by taking the pushout of the two morphisms from the source spec. The morphism between the mediators is not a definitional extension, so A must be extended as well. One way to do this is to make a copy of the pushout as the target spec of the target interpretation. A simpler target with fewer extensions can be found by removing from the copy of the pushout the definitions added by $A \rightarrow S\text{-as-}A$. There may be other definitions that the user will want to remove; such “cleaning up” is a matter of judgment and taste that does not affect the correctness of the result: as long as only defined elements are removed, and none required by the morphism from A , $A \rightarrow A'$ will exist and the pushout will be a definitional extension of it.

Identity completion is an appropriate choice for instantiation of a program scheme. Program schemes extend algorithm theories with sorts and operations needed to express a particular algorithm, such as the natural numbers for iteration control or container sorts such as sets or sequences. The idea is to capture all the problem-dependent aspects of a class of algorithms in the algorithm

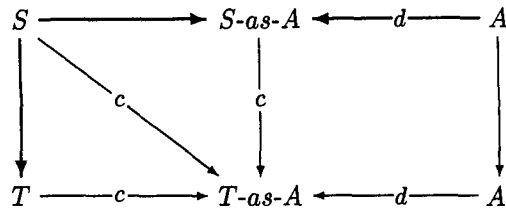


Figure 46. Identity Completion of an Interpretation Morphism

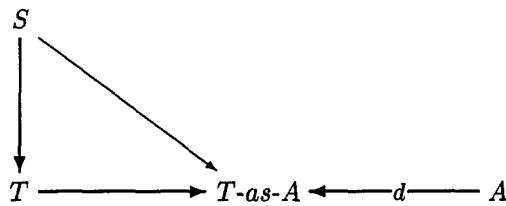


Figure 47. Composing a Morphism with an Interpretation

theory, so that these extensions are completely problem independent. If this is done correctly, then these extensions can be added with the identity completion technique.

In general, the case where the target interpretation is given is easier to solve than when the source interpretation is given. This problem can be approached simply by composition. Figure 47 illustrates this case. Here the morphism $S \rightarrow T$ is simply composed with $T \rightarrow T-as-A$, yielding an interpretation $S \Rightarrow A$. This can be put into the form of an interpretation morphism by using two copies of $T-as-A$ and A and identity morphisms between the copies. The copy of $T-as-A$ in the source interpretation may contain sorts and operations needed for the morphism from T but not in the image of $S \rightarrow T$: these could be removed, if desired, to form a smaller spec $S-as-A$ with morphism $S-as-A \rightarrow T-as-A$. A new target, A' , could then be formed by removing from $S-as-A$ the definitions added by $A \rightarrow T-as-A$, to yield a definitional extension $A' \rightarrow S-as-A$. These clean-up rules do not correspond to any standard categorical construction and require user intervention, but they are also optional.

If $S \rightarrow T$ is a definitional extension, the clean up is more straightforward because you can simply remove from $T-as-A$ those definitions added by $S \rightarrow T$ that are also definitions added by

$A \rightarrow T\text{-as-}A$, so that the morphism between the mediators is a definitional extension, and the new source mediator, $S\text{-as-}A$, is still a definitional extension of A . $S\text{-as-}A$ may still contain elements not needed by the morphism from S , but removing them is more difficult if they are not defined elements, and probably not worth the effort of propagating the changes to A in such a way as to maintain definitional extension. If $S \rightarrow T$ is a translation, one can instead define corresponding translations for the mediator and target, as discussed above. The intent here is to “undo” for $T\text{-as-}A$ and A the translation done to S .

A different technique for completing an interpretation morphism given the target starts by attaching the identity interpretation to S . By composition there is a morphism $S \rightarrow T\text{-as-}A$. There may or may not be one from S to A ; if any of the elements of S were added to A by $T\text{-as-}A$, then they will not be in A . These elements therefore need to be removed from the copy of S that is the target of the source interpretation. This may cause the morphism from it to the mediator copy of S not to be a definitional extension, however. In a diagram refinement context this may not be a problem. A triple of specs with the shape of an interpretation but where the mediator is not a definitional extension of the target is called an *interpretation scheme* (80). An interpretation scheme can be thought of as an interpretation with “holes” in it, meaning those elements in the mediator that are not in the target and not defined. Morphisms between interpretation schemes are defined exactly as for interpretations, and diagram refinement for interpretation schemes is also well-defined. It is possible, in fact, for a diagram of interpretation schemes to have as its colimit an interpretation. This happens when the interpretation schemes overlap in such a way that at least one of them provides a definition for each element that needs one: thus the “holes” in each interpretation scheme are all “filled in” by at least one of the others. In the construction suggested above, the source interpretation scheme that is constructed has holes, but they are all filled in by the target interpretation that was given.

None of the techniques outlined so far is suitable for the classification step of algorithm design, because none of them is capable of integrating the source morphism into the given interpretation in a meaningful way. This problem is arbitrarily difficult and is the essence of design: recognizing in a new problem a familiar structure or pattern. The next section describes a technique that uses stored knowledge about T to support this recognition task.

5.2 *Connections between Specifications*

The concept of *connections between specifications* was invented by Doug Smith to explain certain recurring patterns in the design tactics of KIDS. Formally, a connection defines a set of circumstances under which a given signature morphism is known to preserve theorems and hence to constitute a specification morphism. In practice, the theory of connections provides a constructive technique for completing an interpretation morphism given the source interpretation and an extension of the source spec. This technique relies on a knowledge base of interpretations from the source extension that are reused and adapted to new situations. The only available reference on connections is (72), a mathematically and notationally challenging paper. We will attempt first to provide a gentler introduction to connections. We begin with an example.

5.2.1 A Gentle Introduction to Connections. Figure 48 shows again the spec *Program*, which defines what it means for a program or function F to solve a problem. It extends *Problem* with the operation symbol F and a *correctness axiom* that says that for every valid input, F produces a valid output. Figure 49 graphically portrays the components of *Program* and their relationships. We will use this simple spec to illustrate the key concepts behind connections and how they work.

To simplify the presentation, we will adopt in this section the following conventions: interpretations will be represented as mappings from source names to target expressions rather than with explicit mediator specs, and subscripted names such as I_A will be used to indicate the name

```

spec Program is
  sorts D, R
  op  $I : D \rightarrow \text{Boolean}$ 
  op  $O : D, R \rightarrow \text{Boolean}$ 
  op  $F : D \rightarrow R$ 
  axiom correctness is  $\forall (x : D) (I(x) \Rightarrow O(x, F(x)))$ 
end-spec

```

Figure 48. The Spec *Program*

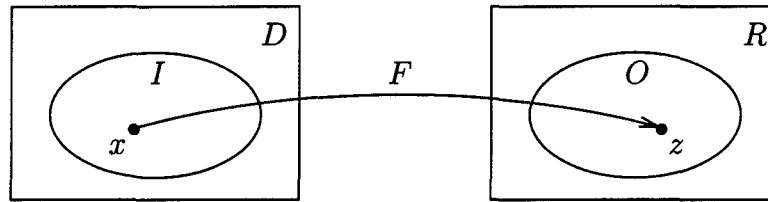


Figure 49. Venn Diagram for *Program Spec*

or expression in spec *A* to which the name *I* is mapped by the interpretation being discussed. In cases where the context is clear, interpretations will be referred to by the name of their target spec, for example as *A* rather than the correct but more cumbersome $\text{Problem} \Rightarrow A$.

Let *A* be a spec in which we have defined a problem to be solved; that is, we have an interpretation from *Problem* to *A*. We wish to find a program to solve this problem; that is, we want an interpretation from *Program* to some extension of *A* that has the same mappings for the components of *Problem*. Referring to Figure 43, we have *Problem* in the role of *S*, *Program* in the role of *T*, *A* as itself, and some mediator *Problem-as-A* that we are not considering directly. Assume further that our knowledge base contains some number of solved problems; that is, some number of interpretations from *Program*. We then ask, under what circumstances can an interpretation from *Program* to some spec *B* be used to solve the problem specified by *A*? To answer this question we will need to combine specs *A* and *B*; this combination will play the role of *A'* in the figure. The details of how they are combined will be presented later; for now we will use the notation $A + B$ to denote the combination. The mediator specs are combined similarly, but this will not be shown explicitly at this time.

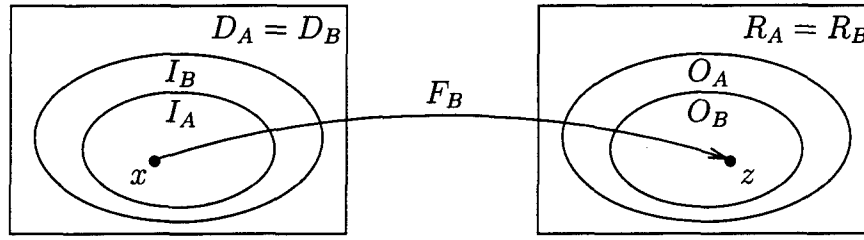


Figure 50. Venn Diagram for Matching an Operator

Let us assume first that $D_A = D_B$ and $R_A = R_B$; that is, the domain and range sorts of A and B are the same once they are combined. Then if the conditions

$$\forall(x : D) (I_A(x) \Leftrightarrow I_B(x))$$

$$\forall(x : D, z : R) (O_A(x, z) \Leftrightarrow O_B(x, z))$$

hold, we already have a solution to A in our library, for B solves precisely the same problem. We shall see, however, that a weaker set of conditions is also sufficient.

Figure 50 shows such a situation. Here we use a Venn diagram to illustrate graphically that the following relationships hold between the operations of A and those of B :

$$\forall(x : D) (I_A(x) \Rightarrow I_B(x)) \tag{3}$$

$$\forall(x : D, z : R) (O_B(x, z) \Rightarrow O_A(x, z)) \tag{4}$$

Under these conditions we can obtain a solution to A by defining

$$F_A(x) = F_B(x)$$

Condition 3 insures that F_B will not be passed an invalid input with respect to spec B ; condition 4 insures that F_B will not produce an invalid output with respect to spec A .

Theorem 5.2.1 *Let $Problem \Rightarrow A$ and $Program \Rightarrow B$ be interpretations to specs A and B such that $D_A = D_B$, $R_A = R_B$ and conditions 3 and 4 above are true. Then the mappings*

$$\begin{aligned} D &\mapsto D_A \\ R &\mapsto R_A \\ I &\mapsto I_A \\ O &\mapsto O_A \\ F &\mapsto \lambda(x : D_A) F_B(x) \end{aligned}$$

define an interpretation $Program \Rightarrow A + B$.

Proof. We need to verify that the axiom of *Program* is a theorem under the mapping indicated.

$$\begin{aligned} I_A(x) & \text{ hypothesis} \\ \Rightarrow I_B(x) & \text{ by condition 3} \\ \Rightarrow O_B(x, F_B(x)) & \text{ by validity of } Program \Rightarrow B \\ \Rightarrow O_A(x, F_B(x)) & \text{ by condition 4} \\ \Leftrightarrow O_A(x, F_A(x)) & \text{ by definition of } F_A \end{aligned}$$

□

By relaxing the requirements that $D_A = D_B$ and $R_A = R_B$ we can find more general conditions for the program of B to solve the problem of A . To do this we introduce *conversion operations*

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_B \rightarrow R_A$$

The conditions describing the required relationship between A and B are modified to incorporate these conversions:

$$\forall(x : D_A) (I_A(x) \Rightarrow I_B(h_D(x))) \tag{5}$$

$$\forall(x : D_A, z : R_B) (O_B(h_D(x), z) \Rightarrow O_A(x, h_R(z))) \quad (6)$$

and we modify the definition of F_A similarly:

$$F_A(x) = h_R(F_B(h_D(x)))$$

This definition accords with the intuitive notion of solving a new problem by converting it to a problem we know how to solve, solving it, and converting the answer back. Under these circumstances, too, we get a valid interpretation.

Theorem 5.2.2 *Let $\text{Problem} \Rightarrow A$ and $\text{Program} \Rightarrow B$ be interpretations to specs A and B and let*

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_B \rightarrow R_A$$

be operations such that conditions 5 and 6 above are true. Then the mappings

$$D \mapsto D_A$$

$$R \mapsto R_A$$

$$I \mapsto I_A$$

$$O \mapsto O_A$$

$$F \mapsto \lambda(x : D_A) h_R(F_B(h_D(x)))$$

define an interpretation $\text{Program} \Rightarrow A + B$.

Proof. We again verify that the axiom of *Program* is a theorem under the mapping indicated.

$$\begin{array}{ll}
I_A(x) & \text{hypothesis} \\
\Rightarrow I_B(h_D(x)) & \text{by condition 5} \\
\Rightarrow O_B(h_D(x), F_B(h_D(x))) & \text{by validity of } Program \Rightarrow B \\
\Rightarrow O_A(x, h_R(F_B(h_D(x)))) & \text{by condition 6} \\
\Leftrightarrow O_A(x, F_A(x)) & \text{by definition of } F_A
\end{array}$$

□

The relationship between B and A is called a *connection from B to A* (with respect to *Program*). We have here a fairly powerful mechanism for reusing knowledge about programs (refinements of spec *Program*) to solve problems (refinements of spec *Problem*). The next subsection presents a general theory of connections that is not dependent on *Problem* or *Program*. It describes formally how the signatures of the conversion operations and the connection conditions can be derived automatically, how the specs A and B are combined, and how the various pieces are used to construct the desired target interpretation and complete the interpretation morphism.

5.2.2 Formal Connection Theory. Figure 51 shows in a diagram how connections are used to complete an interpretation morphism. Inputs to the process are highlighted by the use of thick lines; thinner lines are used for derived components. The overall form of the diagram is a series of refinements of the interpretation given at the top. These refinements are interpretation schemes, meaning not all the elements in the mediator are defined in terms of the target spec, until the bottom refinement is reached. Steps of the refinement are as follows:

1. An initial interpretation scheme $T \Rightarrow A$ is formed syntactically by taking the pushout of $S \rightarrow S\text{-as-}A$ and $S \rightarrow T$, similar to how identity completion is done. The symbol $\oplus_S$ represents a pushout operation with S as the shared part. Smith describes extending a partial signature morphism from T to a complete one by adding “fresh symbols” (72). That is what

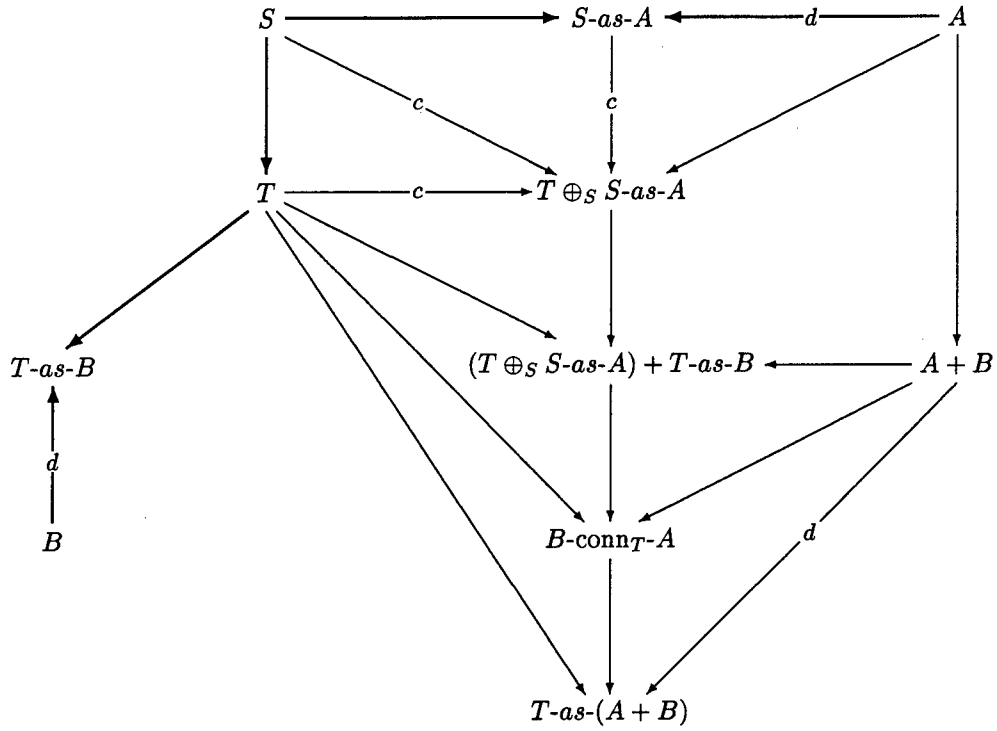


Figure 51. Completing an Interpretation Morphism using Connections

this step accomplishes: $S \rightarrow S\text{-as-}A$ provides the partial mapping and the pushout does the extension. Definitions for the new elements in T are still lacking, but these elements are explicit in the pushout object and can now be manipulated there.

2. An interpretation $T \Rightarrow B$ is selected from a library of refinements of T for the purpose of constructing a connection from it to a refinement of the scheme $T \Rightarrow A$. The constraints imposed by the connection requirements will be used constructively to guide the refinement of the interpretation scheme $T \Rightarrow A$ to a complete interpretation.
3. The two refinements of T , the interpretation $T \Rightarrow B$ and the interpretation scheme $T \Rightarrow A$, are combined. This requires the user to identify any shared parts that they have and then compute a colimit. The details of how they are combined are given below; in this diagram we denote the combination with the $+$ operator.

Note that Figure 51 does not show all the morphisms that exist between objects in the diagram, but only those that commute. So far we have constructed two interpretation scheme morphisms from $S \Rightarrow A$ to insure that the original refinement is preserved at each step of the construction. There are morphisms from $T\text{-as-}B$ to $(T \oplus_S S\text{-as-}A) + T\text{-as-}B$ and from B to $A + B$ that are not shown. By composition, there is a second morphism from T to $(T \oplus_S S\text{-as-}A) + T\text{-as-}B$ that is also not shown: $(T \oplus_S S\text{-as-}A) + T\text{-as-}B$ contains two images of T , one from $T \oplus_S S\text{-as-}A$ and one from $T\text{-as-}B$, but these two images are distinct except for whatever sharing was introduced in the combination. The two refinements of T are emphatically not identified with each other; they are combined into one spec simply so that they can be reasoned about together.

4. The spec $(T \oplus_S S\text{-as-}A) + T\text{-as-}B$ is extended with signatures for conversion operations and with connection conditions as axioms to produce the spec $B\text{-conn}_T\text{-}A$. These signatures and axioms are computed automatically as described below.
5. At this point we have finally assembled in one place all the components we need to derive the definitions that will convert an interpretation scheme into an interpretation. The connection conditions over the elements of S (that is, over the images of these elements in $B\text{-conn}_T\text{-}A$) can be used to derive definitions for the conversion operations, for example by using a constructive theorem proving technique such as unskolemization (72). These connection conditions thus become theorems. With conversions defined, the remaining connection conditions can be used as definitions for the elements that T added to S . Since all the connection conditions are satisfied, the axioms of T are theorems, too. This process transforms $B\text{-conn}_T\text{-}A$ into a new mediator, $T\text{-as-}(A + B)$, that is a definitional extension of $B + A$.

Smith's presentation of connections describes a process whereby a signature morphism is refined into a specification morphism. We have described a process whereby an interpretation scheme is refined into an interpretation. Since SPECWARE has no support for signature morphisms,

it was necessary to insure that only valid specification morphisms were used. The pushout in the first step, for example, insures a morphism $T \rightarrow T \oplus_S S\text{-as-}A$ by the expedient of having in the latter all the axioms of T . By the end of the refinement process these axioms have become theorems, as required for $T\text{-as-}(A + B)$ to be a definitional extension of $A + B$.

When the specs A and B are combined, it is necessary to indicate explicitly if they have any parts in common which are to be identified. The mechanism that SPECWARE provides for combining specs is colimit. The simplest way to combine two specs is to take their coproduct, which is simply the disjoint union of their sorts, operations and axioms. If each spec makes use of the natural numbers, the coproduct will contain two copies of the natural numbers. It is probably more desirable to have only one copy. More importantly, if the intent is that A and B have the same range sort R , say, then this sharing must be made explicit. If this is not done, the connection may not exist or may be very difficult to derive. Figure 52 shows how the sharing between A and B can be identified in a clean and natural way during the connection construction process. Specs A and B are initially combined as a coproduct, which is the colimit of the diagram containing nodes A and B with no arcs. A *shape morphism* is then applied to this diagram (79). A shape morphism maps each spec and morphism in one diagram to itself in another, but the shape of the diagram changes. For instance, we can add a node C and arcs from it to A and B , as shown in the figure. If A and B are structured specs, we can expand their structure as much as necessary in order to reveal the parts that are shared between them; if needed, we can also create new specs to describe the shared parts. There is a unique morphism from $A \oplus B$ to the colimit of this diagram. The mediators $T \oplus_S S\text{-as-}A$ and $T\text{-as-}B$ are also combined as a coproduct, and then the sharing identified between A and B is propagated to the combined mediator by taking the pushout of the two morphisms from $A \oplus B$. This yields an interpretation scheme $T \Rightarrow A \oplus_C B$ from which the connection construction process continues.

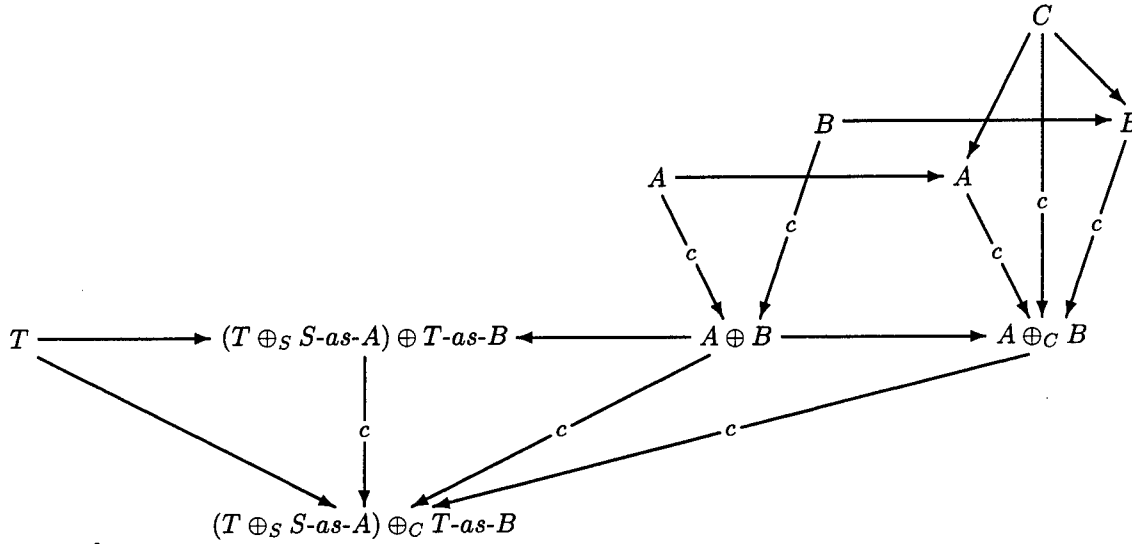


Figure 52. Using a Shape Morphism to Combine Specs

The shape morphism step highlights one of the differences between SPECWARE and other algebraic languages like LSL. In LSL, traits are combined using union instead of colimits. This minimizes duplication in the combined trait, but at the potential cost of unintended identifications via name clashes. To apply connections in a language using union, an explicit renaming step would be required before specs were combined to prevent unwanted sharing. Which method is preferred is largely a matter of taste. Either approach is potentially annoying, but the SPECWARE approach is perhaps safer in that failure to identify sharing will probably lead to difficulty or failure fairly quickly, while unintentional sharing in LSL may go undetected longer. The KIDS implementation of domain theories, not exactly algebraic, is closer in spirit to the Larch style. It is quite convenient to have a rich language like Regroup built into every domain theory, and since domain theories do not have separate name spaces, the compiler detects all name clashes immediately.

5.2.3 Polarity Analysis. The heart of the procedure outlined above is step 4, where the combined mediator is extended with the conversion operations and connection conditions from which will be derived the definitions that refine $T \Rightarrow A$ to an interpretation. The signatures of

the conversion operations (i.e., whether they run from sorts of A to sorts of B or vice versa) and the connection conditions are derived from the axioms of T using *polarity analysis*. This process is explained thoroughly and formally in (72), so it will be described only briefly and informally here. Polarity analysis assigns to each sort and operation of T a *polarity* of $+$, $-$ or $\pm$. These polarity assignments describe how the truth of the axioms of T is affected by changes to its sorts and operations; in effect, we wish to be able to replace the sorts and operations of B with those of A , but only in ways that preserve correctness of the axioms.

Polarities are derived by assigning each axiom of T a polarity of $+$ and recursively propagating polarities through subexpressions until the primitive sort and operation names are reached. Polarity is propagated through an expression according to monotonicity laws describing how the value of the main operation changes as its arguments change. Changes are relative to a partial ordering over the sort in question. Boolean values, for example, can be ordered by implication, and integers by the usual less-than-or-equal relation. Monotonicity laws and polarity analysis rules for the operations of first-order predicate logic and a few others are given in (72). For example, the addition operation is monotonic with respect to both of its arguments, since increasing or decreasing either of them individually increases or decreases their sum. Smith writes the monotonicity law for addition as

$$\text{if } i \xrightarrow[\text{integer}]{p} m \text{ and } j \xrightarrow[\text{integer}]{p} n \text{ then } i + j \xrightarrow[\text{integer}]{p} m + n.$$

Here p is a polarity and $\xrightarrow[\text{integer}]{p}$ represents the interpretation of that polarity for the sort of integers. The law splits into three cases ($p = +, -, \pm$):

$$\text{If } i \leq m \text{ and } j \leq n \text{ then } i + j \leq m + n.$$

$$\text{If } i = m \text{ and } j = n \text{ then } i + j = m + n.$$

$$\text{If } i \geq m \text{ and } j \geq n \text{ then } i + j \geq m + n.$$

This law implies an analysis rule for propagating a polarity through an addition expression,

$$[a + b]^p \Rightarrow a^p + b^p$$

This law says that whenever a sum is assigned a polarity p , both addends are assigned the same polarity. That is, to maintain an ordering between two sums, a sufficient condition is to maintain the same ordering pairwise on the addends.

The subtraction operation is again monotonic with respect to its first argument, but anti-monotonic with respect to its second. Thus when polarity is propagated through a subtraction operation, the polarity of the second argument is inverted. The monotonicity law for subtraction is written

$$\text{if } i \xrightarrow[\text{integer}]{p} m \text{ and } j \xrightarrow[\text{integer}]{\bar{p}} n \text{ then } i - j \xrightarrow[\text{integer}]{p} m - n.$$

which splits into three cases

$$\text{If } i \leq m \text{ and } j \geq n \text{ then } i - j \leq m - n.$$

$$\text{If } i = m \text{ and } j = n \text{ then } i - j = m - n.$$

$$\text{If } i \geq m \text{ and } j \leq n \text{ then } i - j \geq m - n.$$

The corresponding analysis rule is

$$[a - b]^p \Rightarrow a^p - b^{\bar{p}}$$

which says that if a subtraction expression is assigned a polarity p , then the minuend is assigned the same polarity and the subtrahend the inverse polarity. Establishing these orderings on the subexpressions is sufficient to establish the desired ordering on their difference.

Polarity analysis for sorts is very similar. Sorts are ordered by the subsort relation, which is the categorial abstraction of the subset relation. The standard subset relation states that S is a

subset of T if all the elements of S are also elements of T . This implies the existence of an *inclusion* that maps elements of S to themselves in T . More generally, we wish to allow a conversion of some sort to take place between elements of S and T (72). If polarity requires S to be a subsort of T , for example, we require a *conversion operation* from S to a subsort of T . The subsort is not named, but is simply the image of the conversion, which need not be surjective (or *epic*, to use the categorical term).

Polarity analysis rules for sorts are given in a similar notation as for operations. Smith (72) provides three:

$$[\forall(x : D) P(x)]^p \Rightarrow \forall(x : D_{\bar{p}}) P(x)$$

$$[\exists(x : D) P(x)]^p \Rightarrow \forall(x : D_p) P(x)$$

$$[f(a_1, \dots, a_n)]^p \Rightarrow f(a_1, \dots, a_n)_+$$

Intuitively, the first rule corresponds to the monotonicity law that if all the elements of a sort D satisfy a property P , then so do all elements of all subsorts of D but not necessarily all elements of supersorts of D . The second rule corresponds to the law that if sort D contains an element that satisfies P , then so do all supersorts of D but not necessarily all subsorts. The third law says in effect that for any operation $f : S_1, \dots, S_n \rightarrow S$, it is safe to substitute for S a supersort, but not a subsort (since f may return a value not in that subsort).

Smith uses subscripts for sort polarities to distinguish them from operation polarities. We have modified his notation in the first two laws by attaching the polarity to the sort name rather than to the variable x . We do this in order to provide laws not considered by Smith for analyzing the sort constructors of Slang. These constructors are product, coproduct, function, subsort and quotient. Note that the Slang notion of subsort is not the same as the general definition we have been considering.

The following polarity analysis rules for sort constructors are valid in any Slang spec:

$$[S^1, \dots, S^n]_p \Rightarrow S_p^1, \dots, S_p^n$$

$$[S^1 + \dots + S^n]_p \Rightarrow S_p^1 + \dots + S_p^n$$

$$[S \rightarrow T]_p \Rightarrow S_p \rightarrow T_p$$

$$[S \mid P]_p \Rightarrow S_p \mid P^p$$

$$[S/Q]_p \Rightarrow S_p/Q^{\bar{p}}$$

These rules can be understood by relying on our intuitions about sets. In a product sort, substituting a subsort or supersort for any of the components produces a subsort or supersort, respectively, of the product, so polarities are propagated unchanged. Coproducts behave similarly. A function sort is the sort consisting of all possible operations from one sort to another. Substituting subsorts or supersorts of either the domain or codomain yields a subsort or supersort of the function sort.

A Slang subsort $S \mid P$ is identified by a *base sort* S and a predicate $P : S \rightarrow \text{Boolean}$: the subsort consists of all elements of S that satisfy P . If T is a subsort of S (in the general sense), P has a unique restriction to T and $T \mid P$ can be no larger than $S \mid P$. That is, $T \mid P$ is a subsort of $S \mid P$; they are equal only if P is false for all elements of S not in T . If T is a supersort of S and P is extended to T , then regardless of how the extension is done, $S \mid P$ is a subsort of $T \mid P$, and is equal only if P is extended by defining it to be *false* for all elements of T not in S . Thus the base sort S is assigned the same polarity as the subsort expression. The operation P is boolean-valued, so its polarity refers to implication. If a predicate $P' : S \rightarrow \text{Boolean}$ is stronger than P , it is more restrictive and so generates a smaller subsort: $S \mid P'$ is a subsort of $S \mid P$ since P must be *true* whenever P' is. Similarly, if P' is weaker, $S \mid P$ will be a subsort of $S \mid P'$. Thus P also gets the same polarity as the subsort expression. Note that we have assigned an operation polarity during sort polarity analysis.

A Slang quotient sort S/Q is identified by a base sort S and an equivalence relation $Q : S, S \rightarrow \text{Boolean}$: the elements of S/Q are the equivalence classes or orbits of S induced by Q . If T is a subsort of S such that the unique restriction of Q is still an equivalence relation, then T/Q will be a subsort of S/Q : no existing equivalence classes can be split or joined, but only shrunk or deleted. If T is a supersort of S and Q is extended so that it is still an equivalence relation, then S/Q is a subsort of T/Q : the same pairs of elements of S are equivalent as before, so no existing equivalence classes can be split or joined, but only enlarged and new classes added. Thus the base sort S is assigned the same polarity as the quotient sort expression. If Q' is an equivalence relation that is stronger than Q , then it equates fewer pairs of elements, producing more equivalence classes: S/Q is a subsort of S/Q' . If Q' is weaker than Q , then it equates more pairs and produces fewer classes: S/Q' is a subsort of S/Q . Consider the extreme values. If Q' is maximally strong, that is, *false* everywhere, then the equivalence classes of S/Q' all consist of single elements of S : the quotient sort is isomorphic to the base sort, and any weaker equivalence will define a subsort of it. Dually, if Q' is maximally weak, that is, *true* everywhere, then it equates all the elements of S and S/Q' is a sort containing a single element (unless S has no elements); any stronger equivalence will define a supersort of it. Thus Q is assigned the inverse polarity of the quotient expression.

Polarities themselves can be partially ordered, with $\pm$ being less than both $+$ and $-$, which are not ordered with respect to each other. When an element appears more than once in the axioms of T it is assigned more than one polarity. These polarities are combined according to the partial order on polarities by taking their meet. Thus $+$ combined with $-$ is $\pm$, and anything combined with $\pm$ is $\pm$.

A final rule of polarity analysis is that the sort *Boolean* is always assigned a polarity of $\pm$; this will generally be assumed and not shown. This is in accordance with *Boolean* being a built-in sort with a fixed interpretation, namely standard, two-valued predicate logic. Operations built into the logic of Slang, implication, conjunction, quantification and so on, are also assigned $\pm$, implying a

fixed interpretation for these elements as well. A connection will never relate any of these elements to other than themselves.

To illustrate polarity analysis, the axiom of spec *Program* will be analyzed. For sorts, *D* is universally quantified and so gets $-$; *R* appears only as the return value of *F* and so gets $+$. For operations, implication is monotonic in its second argument and antimonotonic in its first, so *I* is assigned $-$ and *O* is assigned $+$. Since we make no assumptions about the monotonicity properties of *O*, *F* is assigned a polarity of $\pm$. To summarize, the polarities assigned to the elements of *Program* are

$$\begin{aligned}\pi_D &= - \\ \pi_R &= + \\ \pi_I &= - \\ \pi_O &= + \\ \pi_F &= \pm\end{aligned}$$

To illustrate the laws for sort constructors, consider an alternative spec for *Program*,

```
spec SubsortProgram is
  sorts D, R
  op I : D → Boolean
  op O : D, R → Boolean
  op F : D | I → R
  axiom correctness is  $\forall(x : D | I) O((\text{relax } I)(x), F(x))$ 
end-spec
```

This spec requires *F* to be defined only for valid inputs, whereas before we required it to be defined everywhere but only specified its behavior for valid inputs. (We could change *O* similarly in *Problem* and *Program* to be defined only for valid inputs.) Polarity analysis of this spec yields the same results as for the standard version. The axiom is universally quantified, so the sort $D | I$ is assigned $-$ and the *O* term is assigned $+$. Both *D* and *I* individually get $-$ by the subsort rule above, *O* is $+$ and again *F* is $\pm$.

Once polarities have been assigned, derivation of the conversion operation signatures and connection conditions is immediate. For a sort *S*, a polarity of $+$ means the sort from *B* must be

converted to a subsort of the one from A , so the conversion operation has signature

$$h_S : S_B \rightarrow S_A$$

A $-$ polarity means the sort from A must be converted to a subsort of the one from B , so the conversion operation has signature

$$h_S : S_A \rightarrow S_B$$

A $\pm$ polarity means the two sorts must be isomorphic, so the conversion operation is a bijection

$$h_S : S_B \leftrightarrow S_A$$

For an operation $f : S_1, \dots, S_n \rightarrow S$ that returns a sort partially ordered by some relation $\preceq$, a polarity of $+$ generates the condition

$$f_B(s_1, \dots, s_n) \preceq f_A(s_1, \dots, s_n)$$

where the conversion operations applied to $s_1, \dots, s_n$ and F and the universal quantifier for the former have been omitted for clarity; a $-$ polarity generates

$$f_B(s_1, \dots, s_n) \succeq f_A(s_1, \dots, s_n)$$

and a $\pm$ polarity generates

$$f_B(s_1, \dots, s_n) = f_A(s_1, \dots, s_n)$$

No conversion operation or connection conditions are generated for the sort *Boolean* and its built-in operations.

Applying these rules to the elements of *Program* according to the polarities derived above yields the conversion operation signatures and connection conditions used in the previous section. Given the situation of Figure 43, Theorem 4.2 of (72) states in effect that whenever conversion operations exist with the signatures shown and such that the connection conditions are satisfied, an interpretation also exists. Theorems 5.2.1 and 5.2.2 above are special cases of this theorem. Figure 51 shows how Theorem 4.2 is applied to complete a partial interpretation morphism, and how A is extended in the process.

5.2.4 Connections in Practice. Connection theory has been applied in the design tactics of KIDS, or rather the former was invented to explain the latter (not to say that all tactics use connections). The theory provides a good accounting of the main task of such tactics, which is of course to complete an interpretation morphism. Most of the other tasks of a tactic, which include instantiating program schemes and algorithm specific tasks such as the filters and cutting constraints of global search (69, 75), are largely orthogonal to this task. The fit between connection theory and how KIDS tactics actually work is not yet perfect, however, and these differences, until accounted for, prevent us from writing a generic mini-tactic for completing interpretation morphisms. This section identifies and explains, as far as is known, the deviations between the theory and practice of connections.

Polarity analysis and the generation of conversion operation signatures and connection conditions are straightforward and could easily be written as an algorithm. The results of this process are not generally used exactly as described above, however. In particular, it is not necessary to analyze all of the axioms of T , nor is it necessary to assign polarities and generate connection conditions for all of the elements of T . The reasons are as follows.

Connections serve to identify a relationship between B and A in terms of their T -structure, yet A does not yet have a fully-identified T -structure. A and B both have S -structure, however, and it is over this structure that the connection must exist. For the diagram of Figure 51 to

commute, it is the elements of S (and A) that must be translated consistently. The first step in the process, forming the interpretation scheme from T to A via pushout, serves only to add fresh names and some axioms to $S\text{-as-}A$ that have no real connection (pardon the pun) yet with A . The only elements of T that we care about, the only ones that can affect the outcome, are the ones that are in the image of $S \rightarrow T$. For the other elements of T , we might just as well use equivalence for operations and isomorphism for sorts: this is the easiest thing to do, and is at least as strong as the corresponding connection conditions or conversion operation signatures. Moreover, using equivalence insures that we have in fact *defined* T -elements for A and not merely constrained them by a weaker axiom. Therefore, we need only calculate polarities and so on for elements in the image of $S \rightarrow T$, and only axioms of T that contain these elements need be included in the polarity analysis. During algorithm design, the role of S is often played by *Problem*, so D , R , I and O are the only elements considered in establishing a connection. Once conversion operations for D and R are found that satisfy the connection conditions for I and O , these conversions are used to define the other elements of T via equality.

An exception to this rule is made for elements that are defined in T . If $S \rightarrow T$ is a definitional extension, then there is no need to build a connection: the pushout operation extends $S \Rightarrow A$ to its unique solution $T \Rightarrow A$. When T extends S by a mixture of defined and undefined elements, the connection mechanism should treat them differently. The pushout operation in step 1 puts defined elements into the pushout with definitions in terms of other elements from T or from $S\text{-as-}A$. There is no need to use a sort axiom or connection condition to define such elements in terms of elements from $T\text{-as-}B$. Moreover, the axioms used to define operations should not be included in the polarity analysis, even if they refer to sorts or operations in S . Operation definitions are a prime source of axioms involving equalities, yielding neutral polarities for the operations involved and hence very strong connection conditions. Because definitions do not restrict the theory generated by a spec but serve only to name existing constructs, the connection conditions for operations involved in the definition do not need to be this strong. Only the element being defined is constrained by the

definition, and this definition is being used as is, so no further consideration of the contents of the definition is required or useful.

On the other hand, defined elements may appear in other axioms that are analyzed and so may be assigned a polarity. For a connection to exist, an operation as defined for A and as defined for B must satisfy the condition required by the polarity for that operation. This should be the actual polarity assigned by analysis, not the default neutral polarity used to define operations not defined in T . For defined sorts, the polarity gives signatures of conversion operations between T sorts as defined for A and those as defined for B . Definitions for these conversions are derived in the same way as for conversions for sorts of S .

In step 2, an interpretation $T \Rightarrow B$ is chosen from a library of refinements of T . In principle one could attempt to establish a connection to a given spec A using every known $T \Rightarrow B$ in the knowledge base, but this would be time consuming and wasteful. The Global Search design tactic of KIDS relies on the user to select the B from which a connection to A is attempted. In order to keep this list short, KIDS makes the simplifying assumption that $R_A = R_B$ even though for global search theory the polarity of R is $-$. This restriction may cause the tactic to miss some opportunities that might have worked, but in practice the risk seems small compared to the benefit. The Operator Match tactic goes further, requiring both $D_A = D_B$ and $R_A = R_B$. This tactic is in other ways more general than connection analysis suggests: it derives sufficient conditions for the connection conditions to hold rather than requiring them to hold always. In this situation, the spec B provides a partial solution to the problem of A rather than a total one. The choices made to keep tactics reasonable or to provide extra capability are thus seen to affect how connection theory is applied.

As a side note, it is not clear how restrictions like those above could be imposed in an environment such as SPECWARE, where expressions like $R_A = R_B$ have no well-defined meaning: elements in separate specs are not comparable in this way. Matching names would be an unreliable

guide to the user's intentions, even as a heuristic. Structural analysis might be better: look at how specs A and B are constructed to see if the elements R_A and R_B are first introduced in references to the same spec. One could require that the combination step identify the sorts R_A and R_B but this step occurs after B has already been selected and thus cannot serve as a filter for selecting it. For the foreseeable future, the user must be relied upon to suggest likely candidates from the entire library of T -refinements.

For the case of algorithm design we can also show that the connection conditions generated as described above are stronger than they need to be. That is, a weaker set of conditions provides the same guarantee of an interpretation and interpretation morphism. The weaker set allows many interpretations to be formed that would be missed if connection theory were followed faithfully. The connection condition that is weakened is the one for O .

In a problem specification, O describes a relationship between inputs and outputs. The only part of interest to a specifier or designer, however, is the relationship between *valid* inputs and their corresponding outputs. This is illustrated by the correctness axiom of *Program*,

$$\forall(x : D) (I(x) \Rightarrow O(x, F(x)))$$

where the output condition is "guarded" by the input condition. The behavior of the program F on invalid inputs is irrelevant. From a design perspective, it is advantageous to leave this behavior unspecified and so allow the designer to make whatever assumptions are desired to produce an efficient program. For the specifier, it is more convenient to assume I holds and focus exclusively on writing O to describe outputs for valid inputs without worrying about what it means otherwise. Since O is assumed to be a total relation¹, we cannot literally leave part of its behavior unspecified. We can, however, ignore what it says concerning invalid inputs. We do this by modifying the

¹The issue of how to deal with partial functions is a complex one that can be resolved in a number of ways. One is to admit them directly and attempt to deal with undefined expressions. Another is to define explicit "error" values that can then be reasoned about. A third is to define a subsort over which the function is total. Each of these approaches is theoretically sound, but none is convenient in practice.

connection condition for O slightly, in a way that still allows an interpretation to be constructed: what seems to work is simply to include $I(x)$ in the antecedent. We will prove that it works for Operator Match.

Theorem 5.2.3 *Let $\text{Problem} \Rightarrow A$ and $\text{Program} \Rightarrow B$ be interpretations to specs A and B and let*

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_B \rightarrow R_A$$

be operations such that condition 5 above and the modified condition

$$\forall(x : D_A, z : R_B) (I_A(x) \wedge O_B(h_D(x), z) \Rightarrow O_A(x, h_R(z))) \quad (7)$$

are true. Then the mappings

$$\begin{aligned} D &\mapsto D_A \\ R &\mapsto R_A \\ I &\mapsto I_A \\ O &\mapsto O_A \\ F &\mapsto \lambda(x : D_A) h_R(F_B(h_D(x))) \end{aligned}$$

define an interpretation $\text{Program} \Rightarrow A + B$.

Proof. The proof is the same as the proof of Theorem 5.2.2 except that step 4 relies on two previous steps rather than just step 3:

$$\begin{array}{ll}
I_A(x) & \text{hypothesis} \\
\Rightarrow I_B(h_D(x)) & \text{by condition 5} \\
\Rightarrow O_B(h_D(x), F_B(h_D(x))) & \text{by validity of } Program \Rightarrow B \\
\Rightarrow O_A(x, h_R(F_B(h_D(x)))) & \text{by condition 7 and hypothesis} \\
\Leftrightarrow O_A(x, F_A(x)) & \text{by definition of } F_A
\end{array}$$

□

Condition 7 says in effect that O_A and O_B are compatible for valid inputs, implying that how they relate with respect to invalid inputs is irrelevant. A modified connection condition is also used in the Global Search tactic, where the O condition derived from the theory is

$$\forall(x : D_A, z : R) (O_A(x, z) \Rightarrow O_B(x, z))$$

(assuming as above that $R_A = R_B$ and using R for this sort), but the tactic uses

$$\forall(x : D_A, z : R) (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(x, z))$$

instead. The proof that this change is valid is very similar to the proof for Operator Match. One can speculate that this technique will work for all algorithm theories, since they are all extensions of *Problem* and used in essentially the same way in a design tactic. It is not at all clear, unfortunately, how to describe or explain what was done in the general case, in order to identify when it is safe or desirable to do this sort of thing.

To summarize, the theory of connections is not quite developed enough to automate the process described in Section 5.2.2. This does not prevent its use; it only requires more of the steps to involve the user than we might otherwise prefer. The remaining section of this chapter

provides an example of how this process is used to classify a problem as suitable for a global search algorithm.

5.2.5 Example Connection. Figure 53 shows a domain theory being built up for the K -Queens problem, a classic combinatorial puzzle. Figure 54 shows the specs. Figure 55 shows the problem specification in canonical form. The input is a positive integer k . The objective is to place k queens on a k -by- k chess board such that no queen can attack any other. That is, no two queens can be placed in the same row, column or diagonal. The output sort is a map from rows to columns, both represented as integers. The spec *Map-I-I* specifies such a map. Two distinct morphisms from *Triv* to *FMap* are needed to identify both the domain and codomain sorts as integers. Using a map representation automatically enforces the row constraint, since it is not possible for a row to be assigned two values by a map. If the map is injective, then the column constraint is enforced as well. Two new predicates, *n2qpud* and *n2qpdd*, are defined to check for violations along diagonals. The names are meant to suggest “no two queens per up (or down) diagonal,” where *up* diagonals have a positive slope when viewed from white’s side of the board and *down* diagonals have a negative slope. For $k \geq 4$, the problem has multiple solutions; the specification given asks for any one of them.

Figure 56 shows a specification for global search theory based mostly on (69) but with some modifications introduced in (75). The differences from Figure 2 are that the operation $\hat{r}_0$ has been renamed *top*, the *Split* operation has been replaced with two separate operations and a new sort, $\hat{C}$, has been introduced. $\hat{C}$ represents *splitting information* that is used when a subspace is split. *Split-Arg* tests whether a splitting variable is valid for a given subspace, and *Split-Constraint* describes the actual splitting operation. A specification for natural numbers, *Nat*, is included so that an explicit definition for *Split** can be provided. *Nat* was defined in Chapter II.

Next we need a library of interpretations from *GlobalSearch* to have something from which to build a connection to K -Queens. Figure 57 is a domain theory and Figure 58 a problem specification

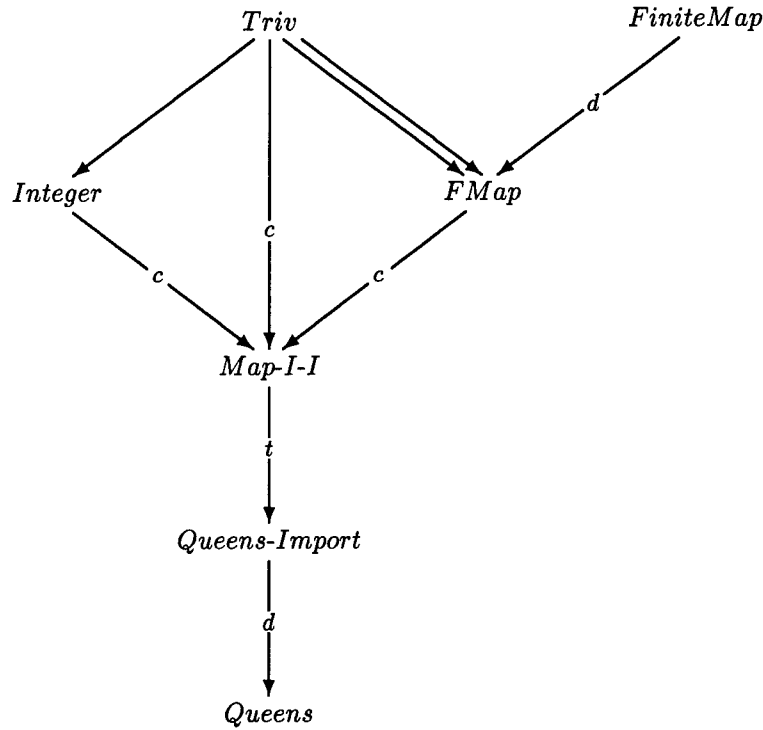


Figure 53. Building a Domain Theory for the K -Queens Problem

for maps over a given domain set into a given range set. This set of maps is the space of potential solutions for the K -Queens problem, and for many other problems with maps as the range sort, such as boolean satisfiability. This is the one-solution form of the problem, asking for any one map from this space. It is named *EnumerateMaps* in anticipation of being used for *searching* the solution space for particular elements. The input sort D is a pair (i.e., a product sort) of sets; pairs are legal inputs if both sets are non-empty.

Figures 59 and 60 extend the problem specification to a global search specification, via an interpretation morphism. The target spec *MapOverSets* is extended by adding *Nat* to it. *Nat* is also added to the mediator *EnumerateMaps*, which is then imported and further extended with the global search sorts and operations. The sort for subspace descriptors is *Map*, the same as the range sort. Legal subspace descriptors are maps whose domain is a subset of the given domain set and whose range is a subset of the given range set. The *Top* descriptor is the empty map, regardless

```

spec FMap is
  import FiniteMap

  op Injective : Map → Boolean
  definition of Injective is
    axiom  $\forall (m : \text{Map}) (\text{Injective}(m) \Leftrightarrow$ 
       $\forall (i : \text{Dom}, j : \text{Dom}, m\text{-at-}i : (\text{Map}, \text{Dom}) \mid \text{defined-at?},$ 
       $m\text{-at-}j : (\text{Map}, \text{Dom}) \mid \text{defined-at?})$ 
       $(\langle m, i \rangle = (\text{relax defined-at?})(m\text{-at-}i)$ 
       $\wedge \langle m, j \rangle = (\text{relax defined-at?})(m\text{-at-}j)$ 
       $\wedge \text{map-apply}(m\text{-at-}i) = \text{map-apply}(m\text{-at-}j) \Rightarrow i = j))$ 
    end-definition
  theorem Injective(empty-map)
end-spec

spec Map-I-I is
  colimit of diagram
    nodes FMap, Integer, Triv
    arcs Triv → Integer : {E → Integer},
      E-COD : Triv → FMap : {E → Cod},
      E-DOM : Triv → FMap : {E → Dom}
  end-diagram

spec Queens-Import is
  translate Map-I-I by {Map → Assignment, E → Integer}

spec Queens is
  import Queens-Import

  op N2QPUD : Assignment → Boolean
  definition of N2QPUD is
    axiom  $\forall (a : \text{Assignment}) (\text{N2QPUD}(a) \Leftrightarrow$ 
       $\forall (i : \text{Integer}, j : \text{Integer}, a\text{-at-}i : (\text{Assignment}, \text{Integer}) \mid \text{defined-at?},$ 
       $a\text{-at-}j : (\text{Assignment}, \text{Integer}) \mid \text{defined-at?})$ 
       $(\langle a, i \rangle = (\text{relax defined-at?})(a\text{-at-}i)$ 
       $\wedge \langle a, j \rangle = (\text{relax defined-at?})(a\text{-at-}j) \wedge i \neq j$ 
       $\Rightarrow \neg(\text{map-apply}(a\text{-at-}i) - i = \text{map-apply}(a\text{-at-}j) - j)))$ 
    end-definition

  op N2QPDD : Assignment → Boolean
  definition of N2QPDD is
    axiom  $\forall (a : \text{Assignment}) (\text{N2QPDD}(a) \Leftrightarrow$ 
       $\forall (i : \text{Integer}, j : \text{Integer}, a\text{-at-}i : (\text{Assignment}, \text{Integer}) \mid \text{defined-at?},$ 
       $a\text{-at-}j : (\text{Assignment}, \text{Integer}) \mid \text{defined-at?})$ 
       $(\langle a, i \rangle = (\text{relax defined-at?})(a\text{-at-}i)$ 
       $\wedge \langle a, j \rangle = (\text{relax defined-at?})(a\text{-at-}j) \wedge i \neq j$ 
       $\Rightarrow \neg(\text{map-apply}(a\text{-at-}i) + i = \text{map-apply}(a\text{-at-}j) + j)))$ 
    end-definition
end-spec

```

Figure 54. Specs for a *K*-Queens Domain Theory

Problem $\longrightarrow$ *QueensProblem* $\longleftarrow_d$ *Queens*

```

spec QueensProblem is
  import Queens

  op I : Integer  $\rightarrow$  Boolean
  definition of I is
    axiom  $\forall(k : \text{Integer}) (I(k) \Leftrightarrow \text{one} \leq k)$ 
  end-definition

  op O : Integer, Assignment  $\rightarrow$  Boolean
  definition of O is
    axiom  $\forall(k : \text{Integer}, a : \text{Assignment}) (O(k, a) \Leftrightarrow$ 
       $\text{Injective}(a) \wedge N2QPUD(a) \wedge N2QPDD(a)$ 
       $\% \text{domain}(a) = \{1..K\}$ 
       $\wedge \forall(j : \text{Integer}) (\text{one} \leq j \wedge j \leq k \Leftrightarrow \text{defined-at?}(a, j))$ 
       $\% \text{range}(a) = \{1..K\}$ 
       $\wedge \forall(a\text{-at-}j : (\text{Assignment}, \text{Integer}) \mid \text{defined-at?})$ 
       $(a = (\text{project } 1)((\text{relax defined-at?})(a\text{-at-}j))$ 
       $\Rightarrow \text{one} \leq \text{map-apply}(a\text{-at-}j) \wedge \text{map-apply}(a\text{-at-}j) \leq k))$ 
    end-definition
end-spec

interpretation K-Queens : Problem  $\Rightarrow$  Queens is
  mediator QueensProblem
  domain-to-mediator  $\{D \rightarrow \text{Integer}, R \rightarrow \text{Assignment}\}$ 
  codomain-to-mediator import-morphism

```

Figure 55. Specification of the *K*-Queens Problem

```

spec GlobalSearch is
  import Problem, Nat

  sorts  $\hat{R}, \hat{C}$ 
  op  $\hat{I} : D, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\text{top} : D \rightarrow \hat{R}$ 
  op  $\text{Satisfies} : R, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\text{Split-Arg} : D, \hat{R}, \hat{C} \rightarrow \text{Boolean}$ 
  op  $\text{Split-Constraint} : D, \hat{R}, \hat{C}, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\text{Split-K} : \text{Nat}, D, \hat{R}, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\text{Split}^* : D, \hat{R}, \hat{R} \rightarrow \text{Boolean}$ 
  op  $\text{Extract} : D, R, \hat{R} \rightarrow \text{Boolean}$ 

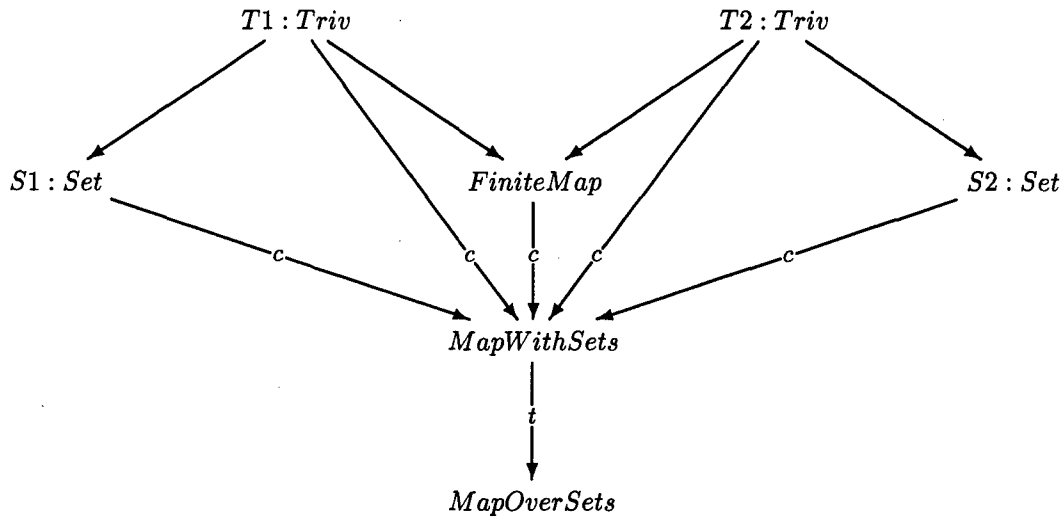
  axiom GS0 is  $\forall(x : D) (I(x) \Rightarrow \hat{I}(x, \text{top}(x)))$ 
  axiom GS1 is  $\forall(x : D, \hat{r} : \hat{R}, \hat{s} : \hat{R})$ 
     $(I(x) \wedge \hat{I}(x, \hat{r}) \wedge \exists(\hat{c} : \hat{C}) (\text{Split-Arg}(x, \hat{r}, \hat{c}) \wedge \text{Split-Constraint}(x, \hat{r}, \hat{c}, \hat{s})) \Rightarrow \hat{I}(x, \hat{s}))$ 
  axiom GS2 is  $\forall(x : D, z : R) (I(x) \wedge O(x, z) \Rightarrow \text{Satisfies}(z, \text{top}(x)))$ 
  axiom GS3 is  $\forall(x : D, z : R, \hat{r} : \hat{R})$ 
     $(I(x) \wedge \hat{I}(x, \hat{r}) \Rightarrow (\text{Satisfies}(z, \hat{r}) \Leftrightarrow \exists(\hat{s} : \hat{R}) (\text{Split}^*(x, \hat{r}, \hat{s}) \wedge \text{Extract}(x, z, \hat{s}))))$ 

  definition of Split-K is
    axiom  $\forall(x : D, \hat{r} : \hat{R}, \hat{s} : \hat{R}) (\text{Split-K}(\text{zero}, x, \hat{r}, \hat{s}) \Leftrightarrow \hat{r} = \hat{s})$ 
    axiom  $\forall(k : \text{Nat}, x : D, \hat{r} : \hat{R}, \hat{s} : \hat{R}) (\text{Split-K}(\text{nat-of-pos}(\text{succ}(k)), x, \hat{r}, \hat{s}) \Leftrightarrow$ 
       $\exists(\hat{c} : \hat{C}, \hat{t} : \hat{R}) (\text{Split-Arg}(x, \hat{r}, \hat{c}) \wedge \text{Split-Constraint}(x, \hat{r}, \hat{c}, \hat{t}) \wedge \text{Split-K}(k, x, \hat{t}, \hat{s})))$ 
    end-definition

  definition of Split* is
    axiom  $\forall(x : D, \hat{r} : \hat{R}, \hat{s} : \hat{R}) (\text{Split}^*(x, \hat{r}, \hat{s}) \Leftrightarrow \exists(k : \text{Nat}) (\text{Split-K}(k, x, \hat{r}, \hat{s})))$ 
    end-definition
end-spec

```

Figure 56. The Global Search Algorithm Theory



spec *MapWithSets* **is**

colimit of diagram

nodes $T1 : Triv, T2 : Triv, S1 : Set, S2 : Set, FiniteMap$

arcs $T1 \rightarrow FiniteMap : \{E \rightarrow Dom\}, T1 \rightarrow S1 : \{\},$
 $T2 \rightarrow FiniteMap : \{E \rightarrow Cod\}, T2 \rightarrow S2 : \{\}$

end-diagram

spec *MapOverSets* **is**

translate *MapWithSets* **by**

$\{Dom \rightarrow Dom, Cod \rightarrow Cod, S1.Set \rightarrow DSet, S1.NE-Set \rightarrow NE-DSet,$
 $S1.delete \rightarrow d-delete, S1.empty-set \rightarrow empty-dset, S1.in \rightarrow d-in,$
 $S1.insert \rightarrow d-insert, S1.nonempty-set? \rightarrow nonempty-dset?,$
 $S1.singleton \rightarrow d-singleton, S1.union \rightarrow d-union, S2.Set \rightarrow CSet,$
 $S2.NE-Set \rightarrow NE-CSet, S2.delete \rightarrow c-delete, S2.empty-set \rightarrow empty-cset,$
 $S2.in \rightarrow c-in, S2.insert \rightarrow c-insert, S2.nonempty-set? \rightarrow nonempty-cset?,$
 $S2.singleton \rightarrow c-singleton, S2.union \rightarrow c-union\}$

Figure 57. Domain Theory for Enumerating Maps

Problem $\longrightarrow$ *EnumerateMaps* $\longleftarrow d$ *MapOverSets*

```

spec EnumerateMaps is
  import MapOverSets

  sort D
  sort-axiom D = DSet, CSet

  op EM-I : D  $\rightarrow$  Boolean
  definition of EM-I is
    axiom  $\forall (U : DSet, V : CSet)$ 
       $(EM-I(\langle U, V \rangle) \Leftrightarrow nonempty-dset?(U) \wedge nonempty-cset?(V))$ 
  end-definition

  op EM-O : D, Map  $\rightarrow$  Boolean
  definition of EM-O is
    axiom  $\forall (U : DSet, V : CSet, M : Map)$   $(EM-O(\langle U, V \rangle, M) \Leftrightarrow$ 
      % Domain valid
       $\forall (x : D) (defined-at?(M, x) \Leftrightarrow d-in(x, U))$ 
      % Range valid
       $\forall (M-at-x : (Map, Dom) \mid defined-at?)$ 
       $(M = (\mathbf{project} \ 1)((\mathbf{relax} \ defined-at?)(M-at-x))$ 
       $\Rightarrow c-in(map-apply(M-at-x, V)))$ 
    end-definition
  end-spec

  interpretation EnumerateMaps : Problem  $\Rightarrow$  MapOverSets is
    mediator EnumerateMaps
    domain-to-mediator  $\{R \rightarrow Map, I \rightarrow EM-I, O \rightarrow EM-O\}$ 
    codomain-to-mediator import-morphism

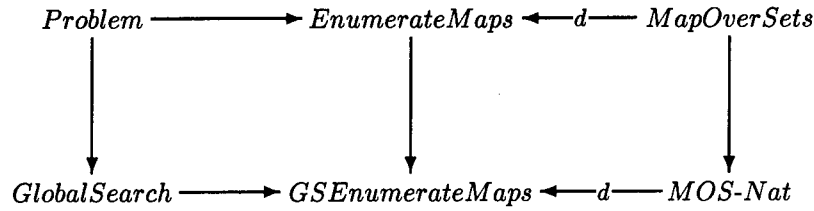
```

Figure 58. Problem Specification for Enumerating Maps

of the input sets. A partial map represents a set of solutions by considering all the ways in which it can be extended; the empty map clearly “contains” all solutions in this sense. A solution belongs to or *Satisfies* a subspace descriptor if it is an extension of it: if the domain of the descriptor is a subset of the domain of the solution, and if the two maps agree over the domain of the descriptor. A space is split by assigning some unassigned domain element a particular range element. As long as the elements are chosen from their respective input sets, legal descriptors are always split into legal descriptors. Once all domain elements have been assigned values, the subspace has become a solution: extraction is simply identity. The operations *Split-K* and *Split** are defined exactly as they are in *GlobalSearch*.

Figures 61 through 64 show the construction of a global search connection from enumerating maps to K-queens. Figure 61 shows the first three steps.

1. An initial interpretation scheme $GSQueens0 : GlobalSearch \Rightarrow Queens$ is formed by computing a pushout. Note that the pushout includes definitions for *Split-K* and *Split**.
2. A global search specification for enumerating maps is selected from a library.
3. The mediator and target specs from steps 1 and 2 are combined. Figure 62 shows the details of the combination. Initially *GSQP-and-GSEM* and *Queens-and-MN* are formed as coproducts of the mediators and targets, respectively. The diagram for *Queens-and-MN* is shown. A shape morphism is applied to this diagram to indicate that the two target specs share an image of *FiniteMap*. The colimit of this diagram is translated to become the combined target spec, *Queens-plus-MN*. The sharing (and renaming) is propagated to the mediator by computing a new mediator as a pushout. Note that the combined mediator contains two images of *Nat*: one from the pushout in step 1 and one from *GSEnumerateMaps* (where it is called *EM-Nat*). Since *Queens* does not have an image of *Nat*, there is no way to indicate sharing in the target diagram and hence no way to identify them in the mediator.



spec *EM-Nat* is
 translate *Nat* by $\{Nat \rightarrow EM-Nat\}$

spec *MOS-Nat* is
 colimit of diagram
 nodes *MapOverSets*, *EM-Nat*
 end-diagram

spec *EnumerateMaps-Nat* is
 colimit of diagram
 nodes *EnumerateMaps*, *EM-Nat*
 end-diagram

interpretation *GSEnumMaps* : *GlobalSearch* $\Rightarrow$ *MOS-Nat* is
 mediator *GSEnumerateMaps*
 domain-to-mediator

$\{R \rightarrow Map, I \rightarrow EM-I, O \rightarrow EM-O, \hat{R} \rightarrow Map, \hat{C} \rightarrow EM-\hat{C},$
 $NAT \rightarrow EM-NAT, \hat{I} \rightarrow EM-\hat{I}, Top \rightarrow EM-Top, Satisfies \rightarrow EM-Satisfies,$
 $Split-Arg \rightarrow EM-Split-Arg, Split-Constraint \rightarrow EM-Split-Constraint,$
 $Split-K \rightarrow EM-Split-K, Split^* \rightarrow EM-Split^*, Extract \rightarrow EM-Extract\}$

codomain-to-mediator $\{\}$

ip-scheme-morphism *EnumMaps-to-GSEnumMaps* : *EnumerateMaps* $\rightarrow$ *GSEnumMaps* is
 domain-sm $\{\}$
 mediator-sm $\{\}$
 codomain-sm $\{\}$

Figure 59. Global Search Applied to Map Enumeration

```

spec GSEnumerateMaps is
  import EnumerateMaps-Nat

  sort EM-Ĉ
  sort-axiom EM-Ĉ = Dom, Cod

  op EM-Î : D, Map → Boolean
  definition of EM-Î is
    axiom  $\forall(U, V, N) (EM-Î(\langle U, V \rangle, N) \Leftrightarrow \forall(x) (defined-at?(N, x) \Rightarrow d-in(x, U))$ 
       $\wedge \forall(N-at-x) (N = (\text{project } 1)((\text{relax } defined-at?)(N-at-x))$ 
       $\Rightarrow c-in(map-apply(N-at-x, V)))$ 

  end-definition

  op EM-Top : D → Map
  definition of EM-Top is axiom  $\forall(x : D) (EM-Top(x) = empty-map)$ 
  end-definition

  op EM-Satisfies : Map, Map → Boolean
  definition of EM-Satisfies is % z "extends" N
    axiom  $EM-Satisfies(z, N) \Leftrightarrow \forall(x) (defined-at?(N, x) \Rightarrow defined-at?(z, x))$ 
       $\wedge \forall(N-at-x, z-at-x) (N = (\text{project } 1)((\text{relax } defined-at?)(N-at-x))$ 
       $\wedge z = (\text{project } 1)((\text{relax } defined-at?)(z-at-x))$ 
       $\Rightarrow map-apply(N-at-x) = map-apply(z-at-x))$ 

  end-definition

  op EM-Split-Arg : D, Map, EM-Ĉ → Boolean
  definition of EM-Split-Arg is
    axiom  $EM-Split-Arg(\langle U, V \rangle, N, \langle a, b \rangle) \Leftrightarrow d-in(a, U) \wedge \neg defined-at?(N, a) \wedge c-in(b, V)$ 
  end-definition

  op EM-Split-Constraint : D, Map, EM-Ĉ, Map → Boolean
  definition of EM-Split-Constraint is
    axiom  $EM-Split-Constraint(x, N, \langle a, b \rangle, M) \Leftrightarrow M = map-shadow(N, a, b)$ 
  end-definition

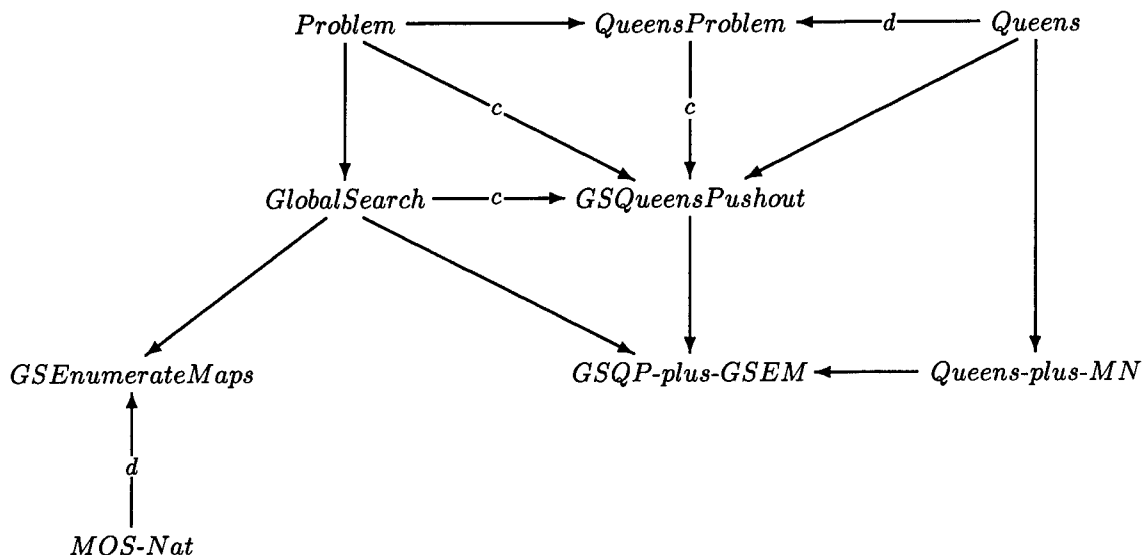
  op EM-Split-K : EM-Nat, D, Map, Map → Boolean
  definition of EM-Split-K is
    axiom  $EM-Split-K(zero, x, \hat{r}, \hat{s}) \Leftrightarrow \hat{r} = \hat{s}$ 
    axiom  $EM-Split-K(succ(k), x, \hat{r}, \hat{s}) \Leftrightarrow$ 
       $\exists(\hat{c}, \hat{t}) (EM-Split-Arg(x, \hat{r}, \hat{c}) \wedge EM-Split-Constraint(x, \hat{r}, \hat{c}, \hat{t}) \wedge EM-Split-K(k, x, \hat{t}, \hat{s}))$ 
  end-definition

  op EM-Split* : D, Map, Map → Boolean
  definition of EM-Split* is axiom  $EM-Split^*(x, \hat{r}, \hat{s}) \Leftrightarrow \exists(k) (EM-Split-K(k, x, \hat{t}, \hat{s}))$ 
  end-definition

  op EM-Extract : D, Map, Map → Boolean
  definition of EM-Extract is
    axiom  $EM-Extract(x, N, M) \Leftrightarrow \forall(a) (defined-at?(N, a) \Leftrightarrow d-in(a, U)) \wedge N = M$ 
  end-definition
end-spec

```

Figure 60. Global Search Applied to Map Enumeration, Cont.

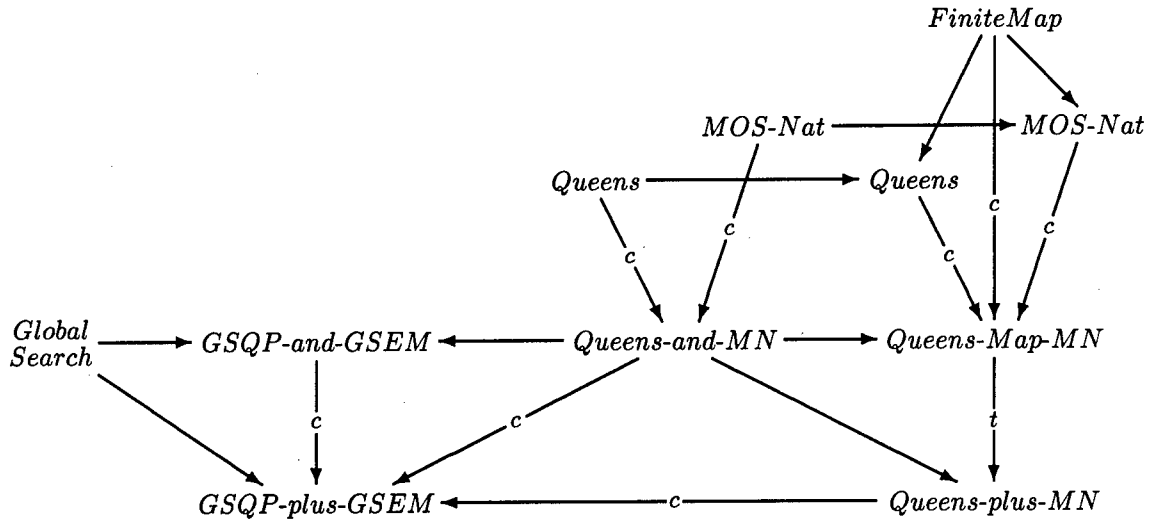


spec *GSQueensPushout* is
 colimit of diagram
 nodes *Problem*, *GlobalSearch*, *QueensProblem*
 arcs *Problem* → *GlobalSearch* : {}
 Problem → *QueensProblem* : {*D* → *Integer*, *R* → *Assignment*}
 end-diagram

ip-scheme *GSQueens0* : *GlobalSearch* ⇒ *Queens* is
 mediator *GSQueensPushout*
 domain-to-mediator cocone-morphism from *GlobalSearch*
 codomain-to-mediator {}

ip-scheme-morphism *Queens-to-GSQueens0* : *K-Queens* → *GSQueens0* is
 domain-sm {}
 mediator-sm cocone-morphism from *QueensProblem*
 codomain-sm identity-morphism

Figure 61. A *GlobalSearch*-Connection for *K*-Queens, Steps 1–3



spec *GSQP-and-GSEM* is
 colimit of diagram nodes *GSQueensPushout*, *GSEnumerateMaps* end-diagram

spec *Queens-and-MN* is
 colimit of diagram nodes *Queens*, *MOS-Nat* end-diagram

spec *Queens-Map-MN* is
 colimit of diagram
 nodes *FiniteMap*, *Queens*, *MOS-Nat*
 arcs *FiniteMap* $\rightarrow$ *Queens* : { *Map* $\rightarrow$ *Assignment*, *Dom* $\rightarrow$ *Integer*, *Cod* $\rightarrow$ *Integer* }
FiniteMap $\rightarrow$ *MOS-Nat* : {}
 end-diagram

spec *Queens-Plus-MN* is
 translate *Queens-Map-MN* by { *Map* $\rightarrow$ *Assignment*, *Dom* $\rightarrow$ *Integer*, *Cod* $\rightarrow$ *Integer* }

spec *GSQP-plus-GSEM* is
 colimit of diagram
 nodes *Queens-and-MN*, *GSQP-and-GSEM*, *Queens-plus-MN*
 arcs *Queens-and-MN* $\rightarrow$ *GSQP-and-GSEM* : {}
Queens-and-MN $\rightarrow$ *Queens-plus-MN* :
 { *Map* $\rightarrow$ *Assignment*, *Dom* $\rightarrow$ *Integer*, *Cod* $\rightarrow$ *Integer* }
 end-diagram

ip-scheme *GSQueens1* : *GlobalSearch* $\Rightarrow$ *Queens-plus-MN* is
 mediator *GSQP-plus-GSEM*
 domain-to-mediator { *D* $\rightarrow$ *Integer*, *R* $\rightarrow$ *Assignment* }
 codomain-to-mediator cocone-morphism from *Queens-plus-MN*

ip-scheme-morphism *Queens-to-GSQueens1* : *K-Queens* $\rightarrow$ *GSQueens1* is
 domain-sm {} mediator-sm {} codomain-sm {}

Figure 62. A *GlobalSearch*-Connection for *K*-Queens, Step 3 Details

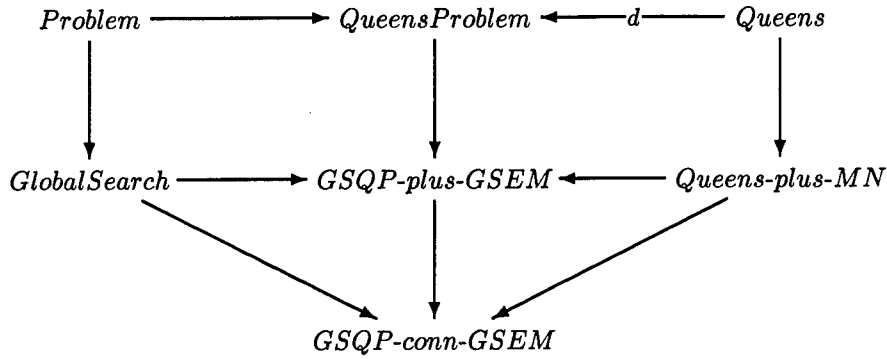
4. Analysis of *GS0–GS3* yields the following polarities for the elements of *GlobalSearch*:

π_D	=	–
π_R	=	–
$\pi_{\hat{R}}$	=	$\pm$
$\pi_{\hat{C}}$	=	+
π_I	=	–
π_O	=	–
π_f	=	$\pm$
π_{top}	=	$\pm$
$\pi_{Satisfies}$	=	$\pm$
$\pi_{Split-Arg}$	=	–
$\pi_{Split-Constraint}$	=	–
π_{Split^*}	=	$\pm$
$\pi_{Extract}$	=	$\pm$

The operation *Split-K* does not appear in the assigned axioms and so is not assigned a polarity. The axioms of *Nat* do not involve any elements of *Problem*, so they are not analyzed and none of the sorts and operations of *Nat* are assigned polarities.

The combined mediator is extended in Figure 63 with conversion operations and connection conditions relating the two images of *GlobalSearch* that it contains. Conversion operations are not needed for *R*, because the output sorts of the two images were identified in the previous step, or for $\hat{R}$ or $\hat{C}$, because we use sort axioms instead to identify them with the corresponding sorts from enumerating maps. The connection conditions for operations from *Problem* are given as axioms; the axiom for *O* is the weaker form discussed above. A condition for *Split** is also given because it is a defined operation. Connection conditions for the other operations of *GlobalSearch* are always given as equalities, regardless of their computed polarities, and presented as definitions. The elements of *Nat* are virtually identified with their counterparts in *EM-Nat*, which we would have identified formally in step 3 had we been able.

5. The connection spec is transformed in Figure 64 into a definitional extension of *Queens-plus-MN* by finding a definition for h_D that satisfies the three connection axioms. For the definition shown, it is easy to see for example that $one \leq k$ implies that the sets returned by h_D will be nonempty. The condition for *Split** follows from the definitions of *Split-Arg*,



spec *GSQP-conn-GSEM* is
 import *GSQP-plus-GSEM*
 sort-axiom $\hat{C} = EM\text{-}\hat{C}$
 sort-axiom $\hat{R} = \text{Assignment}$
 sort-axiom $\text{Nat} = EM\text{-Nat}$
 op $h_D : \text{Integer} \rightarrow D$
 axiom *I-Condition* is $\forall(k : \text{Integer}) (I(k) \Rightarrow EM\text{-}I(h_D(x)))$
 axiom *O-Condition* is $\forall(k : \text{Integer}, a : \text{Assignment}) (I(k) \wedge O(k, a) \Rightarrow EM\text{-}O(h_D(k), a))$
 axiom *Split*-Condition* is
 $\forall(k : \text{Integer}, \hat{r} : \hat{R}, \hat{s} : \hat{R}) (\text{Split}^*(k, \hat{r}, \hat{s}) \Leftrightarrow EM\text{-Split}^*(h_D(x), \hat{r}, \hat{s}))$
 definition of $\hat{I}$ is axiom $\forall(k, N) (\hat{I}(k, N) \Leftrightarrow EM\text{-}\hat{I}(h_D(k), N))$ end-definition
 definition of *Top* is axiom $\forall(k : \text{Integer}) (\text{Top}(k) = EM\text{-Top}(h_D(k)))$ end-definition
 definition of *Satisfies* is
 axiom $\forall(a : \text{Assignment}, N : \text{Assignment}) (\text{Satisfies}(a, N) \Leftrightarrow EM\text{-Satisfies}(a, N))$
 end-definition
 definition of *Split-Arg* is
 axiom $\forall(k, N, \hat{c}) (\text{Split-Arg}(k, N, \hat{c}) \Leftrightarrow EM\text{-Split-Arg}(h_D(k), N, \hat{c}))$ end-definition
 definition of *Split-Constraint* is
 axiom $\forall(k : \text{Integer}, N : \text{Assignment}, \hat{c} : \hat{C}, M : \text{Assignment})$
 $(\text{Split-Constraint}(k, N, \hat{c}, M) \Leftrightarrow EM\text{-Split-Constraint}(h_D(k), N, \hat{c}, M))$
 end-definition
 definition of *Extract* is axiom $\forall(k, a, N) (\text{Extract}(k, a, N) \Leftrightarrow EM\text{-Extract}(h_D(k), a, N))$
 end-definition
 definition of *Nat.zero* is axiom $\text{Nat.zero} = EM\text{-Nat.zero}$ end-definition
 definition of *Nat.succ* is axiom $\forall(n : \text{Nat}) (\text{Nat.succ}(n) = EM\text{-Nat.succ}(n))$
 end-definition
 end-spec
 ip-scheme *GSQueens2* : *GlobalSearch* $\Rightarrow$ *Queens-plus-MN* is
 mediator *GSQP-conn-GSEM*
 domain-to-mediator $\{D \rightarrow \text{Integer}, R \rightarrow \text{Assignment}\}$
 codomain-to-mediator $\{\}$
 ip-scheme-morphism *Queens-to-GSQueens2* : *K-Queens* $\rightarrow$ *GSQueens2* is
 domain-sm $\{\}$ mediator-sm $\{\}$ codomain-sm $\{\}$

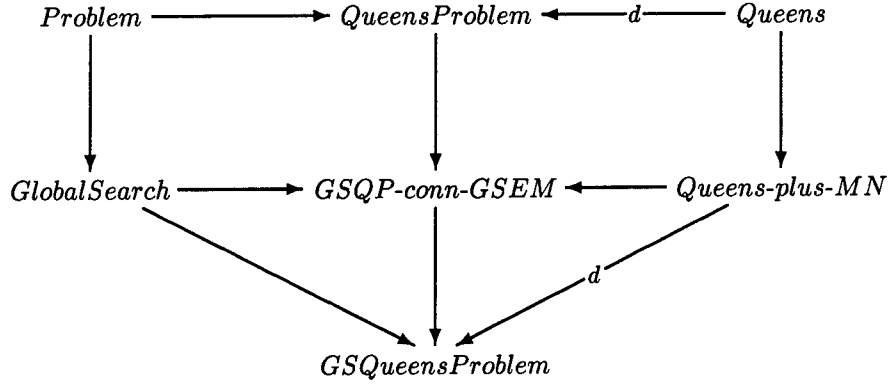
Figure 63. A *GlobalSearch*-Connection for *K*-Queens, Step 4

Split-Constraint and *Split-K*, each in terms of the corresponding operation for enumerating maps. Since the connection axioms now follow logically from the conversion operation definition, these axioms can be dropped (and shown explicitly as theorems, if desired). Likewise the connection conditions insure that the global search axioms are theorems. The transformation is presented as a morphism from the original spec to the final form. Strictly speaking, *GSQueens* does not import *GSQP-plus-GSEM*, which contains the axioms for global search, but it is a convenient fiction in lieu of presenting *GSQueens* as a basic spec.

6. An optional clean-up step could be done to simplify the mediator and target specs. For example, the definition of h_D could be unfolded each place it is used and then eliminated, and the extra copy of *Nat* removed. The definitions of global search operations for enumerating maps could also be unfolded and deleted. The result would be a refinement *GSQueens3* : *GlobalSearch* $\Rightarrow$ *Queens-plus* with an interpretation morphism in the reverse direction of the others, back towards *GSQueens*. The morphism from *GlobalSearch* would insure that all essential structure was maintained. The two interpretation morphisms, from *K-Queens* and *GSQueens2* to *GSQueens*, form a classification diagram. Unlike the diagrams shown in Chapter IV, however, the second interpretation morphism is not composed of definitional extensions.

5.3 Conclusion

This chapter describes a variety of techniques for completing partially constructed interpretation morphisms and gives conditions under which each is an appropriate choice. The algebraic formulation of Smith's connection mechanism is of special importance to algebraic algorithm design, in particular the classification task, because it provides a systematic means for reusing knowledge by adapting it to new situations. Chapter VI formalizes a theory of local search that includes a design tactic for adapting a neighborhood to a problem; this tactic makes use of the connection mechanism. Chapter VI also provides several local search program schemes, which are instantiated



```

spec GSQueensProblem is
  import GSQP-plus-GSEM

  sort-axiom  $\hat{C} = EM-\hat{C}$ 
  sort-axiom  $\hat{R} = Assignment$ 
  sort-axiom  $Nat = EM-Nat$ 

  op  $h_D : Integer \rightarrow D$ 
  definition of  $h_D$  is
    axiom  $\forall(U : DSet, V : CSet, k : Integer) (\langle U, V \rangle = h_D(k) \Leftrightarrow$ 
       $\forall(i : Integer) (d-in(i, U) \Leftrightarrow one \leq i \wedge i \leq k)$ 
       $\wedge \forall(i : Integer) (c-in(i, V) \Leftrightarrow one \leq i \wedge i \leq k))$ 
  end-definition

  theorem I-Condition is  $\forall(k : integer) (I(k) \Rightarrow EM-I(h_D(x)))$ 
  theorem O-Condition is  $\forall(k : Integer, a : Assignment) (I(k) \wedge O(k, a) \Rightarrow EM-O(h_D(k), a))$ 
  theorem Split*-Condition is
     $\forall(k : Integer, \hat{r} : \hat{R}, \hat{s} : \hat{R}) (Split^*(k, \hat{r}, \hat{s}) \Leftrightarrow EM-Split^*(h_D(x), \hat{r}, \hat{s}))$ 
  definition of  $\hat{I}$  is axiom  $\forall(k, N) (\hat{I}(k, N) \Leftrightarrow EM-I(h_D(k), N))$  end-definition
  ...
end-spec

interpretation GSQueens : GlobalSearch  $\Rightarrow$  Queens-plus-MN is
  mediator GSQueensProblem
  domain-to-mediator  $\{D \rightarrow Integer, R \rightarrow Assignment\}$ 
  codomain-to-mediator  $\{$ 

ip-scheme-morphism Queens-to-GSQueens : K-Queens  $\rightarrow$  GSQueens is
  domain-sm  $\{$ 
  mediator-sm  $\{$ 
  codomain-sm  $\{$ 

```

Figure 64. A GlobalSearch-Connection for K -Queens, Step 5

for a particular problem using the identity completion technique. Chapter VII makes further use of the techniques developed in this chapter.

VI. Formalizing Basic Local Search

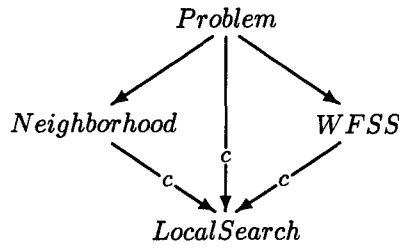
Algorithm classes such as local search are best formalized a step at a time, beginning with the essential, universal properties of the class and then considering various specializations. Figure 65 shows a domain theory for basic local search. This theory extends *WFSS*, the domain theory for optimization problems in general, with a neighborhood structure, represented by the binary relation *N*. This formulation excludes genetic algorithms, but is general enough to encompass all other common forms of local search. The neighborhood structure is independent of the cost function and so it is a separate extension of *Problem*, the space of feasible solutions. The structure of the diagram insures that the neighborhood structure and cost function are defined over the same solution space.

Figure 66 shows a problem specification

$$Problem \longrightarrow LocalOptimum \longleftarrow d \longleftarrow LocalSearch$$

for local optimization. It is analogous to the specification of global optimization given in Chapter IV, except local optimality is relative only to the immediate neighborhood of the solution returned. When local optima are also global optima, we say the neighborhood is *exact* with respect to the cost function (59). Figure 67 shows the spec *LocalOptimum* extended with an axiom defining exactness and a morphism from *GlobalOptimum*, which shows that local search over an exact neighborhood can be used to solve a global optimization problem. For the most part, however, we will not be dealing with exact neighborhoods.

As with the other canonical forms introduced so far, local optimization problems are specified as interpretations from *LocalSearch* to some problem domain. A problem can be specified in this form by the user or classified after the fact. Typically the user will have specified a global optimization problem, defining the feasible solution space and the cost function only. If the user chooses to adopt a local search approach, a neighborhood relation needs to be defined. This neighborhood must be proved exact or the original problem specification must be relaxed to local



```

spec Neighborhood is
  import Problem

  op  $N : D, R, R \rightarrow \text{Boolean}$ 
end-spec

spec LocalSearch is
  colimit of diagram
    nodes Problem, WFSS, Neighborhood
    arcs  $\text{Problem} \rightarrow \text{WFSS} : \{\}$ 
            $\text{Problem} \rightarrow \text{Neighborhood} : \{\}$ 
  end-diagram

```

Figure 65. Local Search Theory

```

spec LocalOptimum is
  import LocalSearch

  op  $\text{LocallyOptimal} : D, R \rightarrow \text{Boolean}$ 
  definition of LocallyOptimal is
    axiom  $\forall (x : D, z : R) (\text{LocallyOptimal}(x, z) \Leftrightarrow$ 
       $\forall (z' : R) (O(x, z') \wedge N(x, z, z') \Rightarrow \text{Cost}(x, z) \leq \text{Cost}(x, z')))$ 
  end-definition

  op  $LO : D, R \rightarrow \text{Boolean}$ 
  definition of LO is
    axiom  $\forall (x : D, z : R) (LO(x, z) \Leftrightarrow O(x, z) \wedge \text{LocallyOptimal}(x, z))$ 
  end-definition
end-spec

interpretation  $\text{LocalOptimization} : \text{Problem} \Rightarrow \text{LocalSearch}$  is
  mediator LocalOptimum
  domain-to-mediator  $\{O \rightarrow LO\}$ 
  codomain-to-mediator import-morphism

```

Figure 66. Problem Specification for Local Optimization

```

spec ExactLO is
  import LocalOptimum

  op Optimal :  $D, R \rightarrow \text{Boolean}$ 
  definition of Optimal is
    axiom  $\forall(x : D, z : R)$ 
       $(\text{Optimal}(x, z) \Leftrightarrow \forall(z' : R) (O(x, z') \Rightarrow \text{Cost}(x, z) \leq \text{Cost}(x, z')))$ 
    end-definition

    axiom exactness is  $\forall(x : D, z : R) (\text{LocallyOptimal}(x, z) \Rightarrow \text{Optimal}(x, z))$ 
  end-spec

morphism GO-to-LO : GlobalOptimum  $\rightarrow$  ExactLO is  $\{GO \rightarrow LO\}$ 

```

Figure 67. Exact Local Search Solves Global Optimization Problems

optimization. Section 6.1 talks in detail about neighborhood structures and their properties, and Section 6.2 presents a tactic for matching neighborhoods to specific problems, completing the classification of the problem. The tactic uses a library of neighborhood specifications and the connection mechanism. Comparisons with Lowry's work are made at appropriate points.

Defining a neighborhood completes the basic classification of the problem as a local search problem. At this point a program scheme can be instantiated or the problem can be further classified to exploit other aspects of its structure. Section 6.3 describe two program schemes, hill climbing and simulated annealing, for which the basic theory of local search is sufficient. Chapter VII extends the basic theory of local search with additional components needed for more advanced techniques.

6.1 Neighborhood Structure

Neighborhood structure is the *sine qua non* of local search algorithms. It describes how new solutions can be generated from old ones and so provides a substrate for the search strategy to build on. Indeed, the terms local search and neighborhood search are used synonymously in the literature.

Figure 68 is a spec for a more detailed theory of neighborhood structure than that given above. This spec provides an internal structure for the neighborhood relation and formalizes several useful

```

spec Neighborhood is
  import Problem

  sort C
  op N_info : D, R, C → Boolean
  op N_is : D, R, C, R → Boolean

  op N : D, R, R → Boolean
  definition of N is
    axiom  $\forall(x : D, z : R, z' : R)$ 
       $(N(x, z, z') \Leftrightarrow \exists(c : C) (N\_info(x, z, c) \wedge N\_is(x, z, c, z')))$ 
  end-definition

  op N* : D, R, R → Boolean
  definition of N* is
    axiom  $\forall(x : D, z : R, z' : R)$ 
       $(N^*(x, z, z') \Leftrightarrow z = z' \vee \exists(z'' : R) (N(x, z, z'') \wedge N^*(x, z'', z')))$ 
  end-definition

  axiom reachability is
     $\forall(x : D, z : R, z' : R) (I(x) \wedge O(x, z) \wedge O(x, z') \Rightarrow N^*(x, z, z'))$ 

  axiom feasibility is
     $\forall(x : D, z : R, z' : R) (I(x) \wedge O(x, z) \wedge N(x, z, z') \Rightarrow O(x, z'))$ 

  axiom perfection is
     $\forall(x : D, z : R, z' : R) (I(x) \wedge O(x, z) \Rightarrow (O(x, z') \Leftrightarrow N^*(x, z, z')))$ 

  axiom symmetry is  $\forall(x : D, z : R, z' : R) (N(x, z, z') \Rightarrow N(x, z', z))$ 

  axiom uniqueness is
     $\forall(x : D, z : R, z' : R, c : C, c' : C)$ 
       $(N\_info(x, z, c) \wedge N\_is(x, z, c, z') \wedge N\_info(x, z, c') \wedge N\_is(x, z, c', z') \Rightarrow c = c')$ 
end-spec

```

Figure 68. Specification of a Theory of Neighborhood Structure

properties. It introduces a new sort, *C*, from which will be taken *local transform values* that will be used in transforming a solution into one of its neighbors. Variables of sort *C* will be called *local transform variables*. The relation *N_info* defines whether a given local transform value is compatible or applicable to a given solution. The relation *N_is* defines when one solution can be transformed into another with a given transform value. The neighborhood relation, *N*, is then defined to hold between two solutions when there exists a legal transform value that transforms the first into the second. The reflexive and transitive closure of *N*, denoted *N**, is also defined.

There are many properties of neighborhoods that are relevant to local search. In the absence of exactness, *reachability* is perhaps the most important one. Reachability asserts that between

any pair of feasible solutions there is a path from one to the other using immediate neighbors. In other words, the graph defined by the reflexive and transitive closure of the neighborhood relation is complete. This property makes it possible, at least in principle, to search the entire space of solutions and so find the optimal one. In practice the space is searched very selectively, but the ability to sample the entire space in one search remains important. Since "true" and many other, non-trivial predicates define reachable neighborhoods not only among feasible solutions but infeasible ones as well, it is important to find reachable neighborhoods that filter out infeasible solutions as completely as possible.

A neighborhood is called *feasible* when it preserves O , the output condition that defines feasible solutions. This property insures that all neighbors of a feasible solution are also feasible. Just as reachable neighborhoods can be too "loose" by allowing infeasible solutions to be neighbors of feasible ones, so too can feasible neighborhoods be too "tight" by violating reachability. The ultimate feasible but unreachable neighborhood is the predicate "false."

When a neighborhood is both reachable and feasible, it is called a *perfect* neighborhood. Perfection is shown in Figure 68 in a more compact form as a separate axiom. As the name suggests, perfection is highly desirable. Most "typical" examples of local search are over perfect neighborhoods.

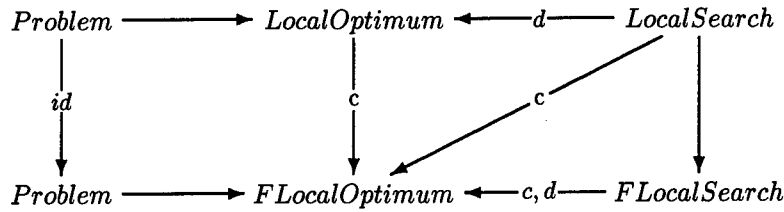
Symmetry is a common but less important property of neighborhoods. Its chief consequence is that when symmetry holds, any move made during local search can be undone in a single move. This sort of property is relevant for local search techniques such as tabu search.

Uniqueness states that between two neighboring solutions there is a unique transform value. Uniqueness is virtually always satisfied and will be assumed unless stated otherwise. A neighborhood that is not unique allows two or more distinct moves between the same pair of solutions, which is not a desirable property during search: it is inefficient if nothing else. Such a neighborhood can

generally be made unique by altering the definition of the sort *C*, for example by using a quotient sort to group together equivalent transform values, or by adding conditions to *N.info*.

Figure 68 shows all these properties in one spec. In practice, it is useful to have separate specs with different combinations of properties, because for many problems the neighborhoods chosen will not possess all of them. Much of the variety among local search algorithms is motivated by the desire to use local search in the absence of one or more of these properties. Even the most desirable properties are optional. If a neighborhood is not reachable, for example, the practice of making repeated searches from independent starts mitigates the impact, though if the space is too fragmented the search still will not be very effective. If a neighborhood is not feasible a number of options are available, including reformulating some feasibility constraints as costs, changing the definition of feasibility (i.e., accepting infeasible solutions directly), finding a way to repair infeasible solutions, and so on. While all these approaches help to make up for the absence of desirable neighborhood properties, most of them are heuristics that can result in inferior solutions. These techniques also increase the complexity of algorithms. Thus it is important to find neighborhoods with as many “good” properties as possible. Different tradeoffs are possible, but in general one sacrifices feasibility if necessary to maintain reachability.

In what follows, the name *Neighborhood* will refer to a spec that defines all the operators described above but without any additional axioms, and the name *LocalSearch* will refer to the colimit of this spec with *Problem* and *WFSS*. These can be used for constructions that do not rely on any particular neighborhood properties. When a property is required, names like *FeasibleNeighborhood* and *FLocalSearch* will be used. Figure 69 shows how to extend the problem class *LocalOptimization* to neighborhoods with properties, using feasibility as an example. First the domain theory is extended by adding one or more neighborhood properties, then this extension is composed with local optimization to form a new problem class and an interpretation morphism from the original one. There are morphisms from specs and interpretations that have fewer properties



spec *FLocalSearch* is

colimit of diagram

nodes *Problem*, *WFSS*, *FeasibleNeighborhood*

arcs *Problem* → *WFSS* : {}

Problem → *FeasibleNeighborhood* : {}

end-diagram

spec *FLocalOptimum* is

colimit of diagram

nodes *LocalSearch*, *LocalOptimum*, *FLocalSearch*

arcs *LocalSearch* → *LocalOptimum* : **import-morphism**

LocalSearch → *FLocalSearch* : {}

end-diagram

interpretation *FLocalOptimization* : *Problem* ⇒ *FLocalSearch* is

mediator *FLocalOptimum*

domain-to-mediator {*O* → *LO*}

codomain-to-mediator **cocone-morphism** from *FLocalSearch*

ip-scheme-morphism *LO-to-FLO* : *LocalOptimization* → *FLocalOptimization* is

domain-sm **identity-morphism**

mediator-sm **cocone-morphism** from *LocalOptimum*

codomain-sm {}

Figure 69. Extending Local Optimization to a Feasible Neighborhood

to those that have more, so whenever a morphism from a particular object is required (such as a reachable neighborhood, say), a morphism from an object with more properties (such as a perfect neighborhood) can safely be used.

Other desirable properties of neighborhoods are not described by the theory, either because they are hard to characterize formally, hard to prove true of specific neighborhoods, or both. Perhaps the most important omitted property is that neighboring solutions are highly similar to each other in a structural sense. The intuition behind a neighborhood structure is that one solution is transformed into another by making some relatively small change to it. The transform variables

act as parameters for this change. Most of the solution is unaffected. A formal theory of structural similarity would probably be based on analysis of the constructors or observers of the solution sort R , but beyond this little is known. Other properties omitted from the theory above include having neighborhoods that are relatively small and uniform in size, and that increase in size at a reasonable rate as problem instance size increases. It is worth noting that neighborhoods based on structural similarity tend to have these nice properties as well.

Neighborhood size also relates to the exactness of local search. Clearly if $N = N^*$, so that every solution is a direct neighbor of every other (and assuming the neighborhood is reachable), then searching this neighborhood is exact but identical to total enumeration and its kin, such as global search. Other neighborhood structures may be smaller but still too large to search effectively. For most problems, even the smallest exact neighborhood is not of reasonable size (59). Even so, large neighborhoods sample a larger proportion of the solution space at each iteration and hence produce better local optima than small neighborhoods, while small neighborhoods take little time to search. Finding an effective balance that produces good optima in reasonable time remains an art. Formal results concerning the computational complexity of finding local optima in general or for particular problems are rare and hard to prove. Three examples should suffice.

For a large class of "reasonable" problems (where costs, neighborhoods and initial solutions can all be computed in polynomial time), finding local optima can be proved to be intermediate in complexity between the problem classes P and NP , but whether it is equal to either (or both) is an open question (42). Interestingly, the authors point out that for some problems one can construct a locally optimal solution with respect to some neighborhood directly rather than by local search; such situations seem rare, however. More recent work draws a connection between polynomial-time local search and the existence of polynomial-time approximation schemes (3). Unfortunately, unless $P = NP$, many problems possess neither (19, 59).

Grover identifies several NP-complete problems that share a property concerning the costs of local optima (33). His conclusions are weak but reassuring: when this property holds, all local optima are of average cost or better, so there are no truly terrible local optima, and from any initial solution one can hill-climb to a better-than-average solution in $O(nL)$ time, where n is the problem instance size, $2^L \geq \bar{C}$ and $\bar{C}$ is the average solution cost (that is, L is any upper bound of $\log_2 \bar{C}$). Note, however, that this better-than-average solution is not a local optimum.

Results like these illustrate the difficulty of analyzing local search algorithms in general: in spite of working surprisingly well in practice, they as often as not defy theoretical analysis. It does not seem worth the effort to incorporate such results in a theory of neighborhood structure or local search such as we are considering here. They may provide some guarantees of performance of an algorithm once it has been developed, but they do not help design it. In practice, performance is usually determined empirically for a given algorithm on a particular problem. The operations research literature is full of papers whose main purpose is to report experiments of this sort ((1, 11, 39, 40, 41, 47, 48, 56, 66), to name but a few).

Very special neighborhood properties such as lattice structure and monotonicity with respect to the cost function lead to specialized algorithms such as fixpoint computation (7, 75). These have not been incorporated into this theory of neighborhoods and local search yet. Possible alternatives to this theory of neighborhood structure include using higher-order relations. These could, for example, describe the cross-over operators used in genetic search.

An example neighborhood will help make this discussion of properties more concrete. The *k-subset-1-exchange* neighborhood is characteristic of a number of problems where search is done over the subsets of a set S that are of size k . Figure 70 presents a problem specification for this space of feasible solutions. The domain theory is an extension of *Set* that has some additional operators. The input is a natural number k and a set S (the elements of which belong to some unspecified sort E), such that the size of S is at least k . The output sort is a set of elements of

sort E and such a set is a feasible solution if it is a subset of S and has size k . Figure 71 then defines a perfect, symmetric neighborhood specification over this space. The transform variables for generating the neighbors of a solution A are a pair of elements of sort E . A pair (i, j) is valid for A whenever i is in S but not in A and j is in A . A new solution is generated from A by inserting i and deleting j .

The following theorem verifies the perfection part of the morphism from *PerfectSymmetricNeighborhood* to *k-Subset-1-Exchange*.

Theorem 6.1.1 *The neighborhood k-subset-1-exchange is perfect.*

Proof. Perfection consists in being both feasible and reachable. We will prove these properties separately.

Feasibility. The feasibility axiom for this neighborhood, with N expanded by its definition, is

$$\begin{aligned}
& \forall(k : \text{Nat}, S : \text{Set}, A : \text{Set}, \text{new\_}A : \text{Set}) \\
& (k \leq \text{size}(S) \wedge k = \text{size}(A) \wedge \text{subsetq}(A, S) \wedge \exists(i : E, j : E) (\text{in}(i, S) \wedge \neg \text{in}(i, A) \\
& \quad \wedge \text{in}(j, A) \wedge \text{new\_}A = \text{insert}(i, \text{delete}(j, A))) \\
& \Rightarrow k = \text{size}(\text{new\_}A) \wedge \text{subsetq}(\text{new\_}A, S))
\end{aligned}$$

Let k, S, A and $\text{new\_}A$ satisfy the antecedent. Since i is not in A and j is, $\text{size}(\text{new\_}A) = \text{size}(A) = k$. Since i is in S and $\text{subsetq}(A, S)$, $\text{subsetq}(\text{new\_}A, S)$.

Reachability. The reachability axiom for this neighborhood is

$$\begin{aligned}
& \forall(k : \text{Nat}, S : \text{Set}, A : \text{Set}, A' : \text{Set}) \\
& (k \leq \text{size}(S) \wedge k = \text{size}(A) \wedge \text{subsetq}(A, S) \wedge k = \text{size}(A') \wedge \text{subsetq}(A', S) \\
& \Rightarrow N_{k\text{slc}}^*(k, S, A, A'))
\end{aligned}$$

$$\text{Problem} \longrightarrow k\text{-Subset} \xleftarrow{d} \text{ExtSet}$$

```

spec ExtSet is
  import Set, Nat

  op subseteq : Set, Set → Boolean
  definition of subseteq is
    axiom  $\forall(S : \text{Set}, T : \text{Set}) (subseteqq(S, T) \Leftrightarrow \forall(x : E) (in(x, S) \Rightarrow in(x, T)))$ 
  end-definition

  op size : Set → Nat
  definition of size is
    axiom  $size(empty\text{-}set) = zero$ 
    axiom  $\forall(x : E, S : \text{Set}) (in(x, S) \Rightarrow size(insert(x, S)) = size(S))$ 
    axiom  $\forall(x : E, S : \text{Set}) (\neg in(x, S) \Rightarrow size(insert(x, S)) = nat\text{-}of\text{-}pos(succ(size(S))))$ 
  end-definition

  theorem  $\forall(x : E, S : \text{Set}) (in(x, S) \Rightarrow succ(size(delete(x, S))) = size(S))$ 
  theorem  $\forall(x : E, S : \text{Set}) (\neg in(x, S) \Rightarrow size(delete(x, S)) = size(S))$ 
  theorem  $\forall(S : NE\text{-}Set) nonzero?(size((relax\ nonempty\text{-}set?)(S)))$ 
end-spec

spec k-Subset is
  import ExtSet

  sort KS-D
  sort-axiom KS-D = Nat, Set

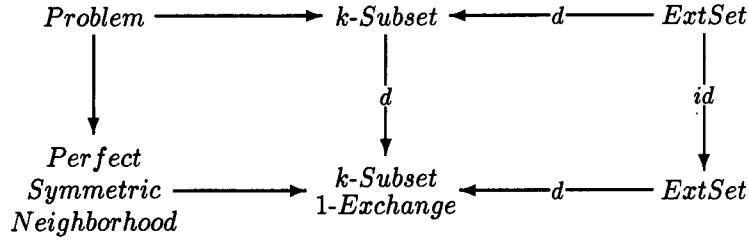
  op KS-I : KS-D → Boolean
  definition of KS-I is
    axiom  $\forall(k : Nat, S : \text{Set}) (KS-I(k, S) \Leftrightarrow k \leq size(S))$ 
  end-definition

  op KS-O : KS-D, Set → Boolean
  definition of KS-O is
    axiom  $\forall(k : Nat, S : \text{Set}, A : \text{Set}) (KS-O(\langle k, S \rangle, A) \Leftrightarrow k = size(A) \wedge subseteq(A, S))$ 
  end-definition
end-spec

interpretation k-Subset : Problem  $\Rightarrow$  ExtSet is
  mediator k-Subset
  domain-to-mediator {D → KS-D, I → KS-I, R → Set, O → KS-O}
  codomain-to-mediator import-morphism

```

Figure 70. Specification of k -Subset Solution Space



spec *k-Subset-1-Exchange* **is**

import *k-Subset*

sort *KS-C*

sort-axiom *KS-C* = *E, E*

op *Exchange_info* : *KS-D, Set, KS-C* → *Boolean*

definition of *Exchange_info* **is**

axiom $\forall (k : \text{Nat}, S : \text{Set}, A : \text{Set}, i : E, j : E)$

$(\text{Exchange\_info}(\langle k, S \rangle, A, \langle i, j \rangle) \Leftrightarrow \text{in}(i, S) \wedge \neg \text{in}(i, A) \wedge \text{in}(j, A))$

end-definition

op *Exchange* : *KS-D, Set, KS-C, Set* → *Boolean*

definition of *Exchange* **is**

axiom $\forall (k : \text{Nat}, S : \text{Set}, A : \text{Set}, i : E, j : E, A' : \text{Set})$

$(\text{Exchange}(\langle k, S \rangle, A, \langle i, j \rangle, A') \Leftrightarrow A' = \text{insert}(i, \text{delete}(j, A)))$

end-definition

op *KS-N* : *KS-D, Set, KS-C* → *Boolean*

definition of *KS-N* **is**

axiom $\forall (x : \text{KS-D}, z : \text{Set}, z' : \text{Set})$

$(\text{KS-N}(x, z, z') \Leftrightarrow \exists (c : \text{KS-C}) (\text{Exchange\_info}(x, z, c) \wedge \text{Exchange}(x, z, c, z')))$

end-definition

op *KS-N** : *KS-D, Set, KS-C* → *Boolean*

definition of *KS-N** **is**

axiom $\forall (x : \text{KS-D}, z : \text{Set}, z' : \text{Set})$

$(\text{KS-N}^*(x, z, z') \Leftrightarrow z = z' \vee \exists (z'' : \text{Set}) (\text{KS-N}(x, z, z'') \wedge \text{KS-N}^*(x, z'', z')))$

end-definition

end-spec

interpretation *ks1e* : *PerfectSymmetricNeighborhood* ⇒ *ExtSet* **is**

mediator *k-Subset-1-Exchange*

domain-to-mediator $\{D \rightarrow \text{KS-D}, I \rightarrow \text{KS-I}, R \rightarrow \text{Set}, O \rightarrow \text{KS-O}, C \rightarrow \text{KS-C},$
 $N\_info \rightarrow \text{Exchange\_info}, N\_is \rightarrow \text{Exchange}, N \rightarrow \text{KS-N}, N^* \rightarrow \text{KS-N}^*\}$

codomain-to-mediator $\{\}$

ip-scheme-morphism *ks1e* : *k-Subset* → *ks1e* **is**

domain-sm **import-morphism**

mediator-sm **import-morphism**

codomain-sm **identity-morphism**

Figure 71. *k-Subset-1-Exchange* Neighborhood

where N_{ks1e}^* is defined as in Figure 68. Let k , S , A and A' satisfy the antecedent. Let U be the symmetric difference of A and A' , $U = (A \setminus A') \cup (A' \setminus A)$. If $size(U) = 0$, then $A = A'$ and the theorem holds by the first part of the definition of N_{ks1e}^* . Now assume the theorem holds whenever $size(U) < n$ and that $size(U) = n$. Let i be an element of $A' \setminus A$ and j an element of $A \setminus A'$, and let $A'' = insert(i, delete(j, A))$. Then A and A'' are neighbors, and the symmetric difference of A'' and A' is $n - 2$. By hypothesis $N_{ks1e}^*(k, S, A'', A')$ and so by the second part of the definition of N_{ks1e}^* , $N_{ks1e}^*(k, S, A, A')$. $\square$

The reachability proof has the form of an induction over a metric defining the distance between two solutions, where the neighborhood structure is used to reduce the distance. This technique is applicable to proving reachability for other neighborhoods: we have applied it to a total of 14. It is not, of course, the only way to proceed, but has been generally useful.

The k -subset-1-exchange neighborhood is also symmetric: the move (i, j) is undone by the move (j, i) . Neighboring solutions are structurally similar, differing only in the placement of one pair of elements of S . If S contains n elements, then each solution has $k(n - k)$ neighbors, which grows polynomially in both n and k . Finally, this neighborhood is unique.

6.2 Neighborhood Matching Tactic

If a problem is specified using the *GlobalOptimization* refinement of *Problem*, then the main task in applying local search to it is to refine the specification of the feasibility problem to a neighborhood specification, that is, an interpretation from the spec *Neighborhood* to (an extension of) the domain theory spec. This refinement can then be combined with the cost function to give a complete interpretation from *LocalSearch*.

Given the difficulty of characterizing many important and desirable properties of neighborhoods such as structural similarity, we should not expect to have a specialized technique for generating neighborhood specifications that guarantees these properties. Indeed, the problem of devising

good neighborhoods for arbitrary problems *ex nihilo* has been closely examined for a long time and remains unsolved (14, 59). The one general mechanism we have for completing interpretation morphisms is connections between specifications. The need for a library of neighborhood specifications, which may seem a weakness of the connection method, in this case gives us the opportunity to hand-craft a set that meets all of our formal criteria and our heuristic and aesthetic ones as well. Appendix A presents such a library for several common structured or composite types (sets, sequences, maps). All of the neighborhoods in the library are perfect unless noted otherwise.

6.2.1 Applying Connections to Neighborhood. To apply connection theory, we need to analyze the axioms of *Neighborhood*, paying particular attention to the sorts D and R and the operations I and O . Depending on which axioms we analyze, we get different connection conditions and correspondingly different property guarantees. As explained in Section 5.2.4, the axioms that define N and N^* are not included in these analyses: as equivalences, they would make many polarities neutral. If these definitions are ignored altogether, however, polarities for N_info , N_is and C will never be assigned. Since these are all elements of the spec playing the role of T in the connection and so destined to be defined by equality regardless of their polarity, this might seem acceptable, if a bit odd. Since the definition of N is so simple, it is possible instead to replace each occurrence of N with its definition so as to assign polarities to these elements. That is, the polarity assigned to N can be assigned to the expression defining N and the analysis continued. This can be also be done for N^* to assign a polarity for N (and hence the others). N^* has a recursive definition, so this analysis assigns N^* a new polarity. In this case the new polarity equals the old one, so the polarity of N^* is not necessarily neutral.

Analyzing the reachability axiom yields the following polarity assignments:

$$\begin{aligned}
\pi_D &= - \\
\pi_R &= \pm \\
\pi_C &= + \\
\pi_I &= - \\
\pi_O &= - \\
\pi_{N\_info} &= + \\
\pi_{N\_is} &= + \\
\pi_N &= + \\
\pi_{N^*} &= +
\end{aligned}$$

The corresponding conversion operations and connection conditions are

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_A \leftrightarrow R_B \quad (\text{assumed to be identity})$$

$$h_C : C_B \rightarrow C_A$$

$$\forall(x : D_A) (I_A(x) \Rightarrow I_B(h_D(x)))$$

$$\forall(x : D_A, z : R) (O_A(x, z) \Rightarrow O_B(h_D(x), z))$$

$$\forall(x : D_A, z : R, c : C_B) (N\_info_B(h_D(x), z, c) \Rightarrow N\_info_A(x, z, h_C(c)))$$

$$\forall(x : D_A, z : R, c : C_B, z' : R) (N\_is_B(h_D(x), z, c, z') \Rightarrow N\_is_A(x, z, h_C(c), z'))$$

$$\forall(x : D_A, z : R, z' : R) (N_B(h_D(x), z, z') \Rightarrow N_A(x, z, z'))$$

$$\forall(x : D_A, z : R, z' : R) (N_B^*(h_D(x), z, z') \Rightarrow N_A^*(x, z, z'))$$

Following (72), and as the comment indicates, the isomorphism between R_A and R_B is assumed to be the identity operation, allowing h_R to be omitted from the connection conditions. This corresponds in SPECWARE to identifying R_A and R_B in the shared part of $A + B$ as being the same sort. This assumption is not necessary in practice, but it often holds and it simplifies the presentation of the connection. If R_A and R_B are distinct but isomorphic sorts, two conversion operations exist, each the inverse of the other, and the connection conditions can be written using either without loss of generality.

Analysis of the feasibility axiom yields the following polarity assignments:

$$\begin{aligned}
 \pi_D &= - \\
 \pi_R &= - \\
 \pi_C &= + \\
 \pi_I &= - \\
 \pi_O &= \pm \\
 \pi_{N\_info} &= - \\
 \pi_{N\_is} &= - \\
 \pi_N &= -
 \end{aligned}$$

and the conversion operations and connection conditions for a feasible neighborhood are

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_A \rightarrow R_B$$

$$h_C : C_B \rightarrow C_A$$

$$\forall(x : D_A) (I_A(x) \Rightarrow I_B(h_D(x)))$$

$$\forall(x : D_A, z : R_A) (O_A(x, z) \Leftrightarrow O_B(h_D(x), h_R(z)))$$

$$\forall(x : D_A, z : R_A, c : C_B) (N\_info_A(x, z, h_C(c)) \Rightarrow N\_info_B(h_D(x), h_R(z), c))$$

$$\forall(x : D_A, z : R_A, c : C_B, z' : R_A)$$

$$(N\_is_A(x, z, h_C(c), z') \Rightarrow N\_is_B(h_D(x), h_R(z), c, h_R(z'))))$$

$$\forall(x : D_A, z : R_A, z' : R_A) (N_A(x, z, z') \Rightarrow N_B(h_D(x), h_R(z), h_R(z')))$$

The axiom for perfection would be expected to yield polarities that combine the above two sets, and this expectation is born out. Analysis of this axiom yields

$$\begin{aligned}
 \pi_D &= - \\
 \pi_R &= - \\
 \pi_C &= + \\
 \pi_I &= - \\
 \pi_O &= \pm \\
 \pi_{N\_info} &= \pm \\
 \pi_{N\_is} &= \pm \\
 \pi_N &= \pm \\
 \pi_{N^*} &= \pm
 \end{aligned}$$

and the conversion operations and connection conditions are

$$h_D : D_A \rightarrow D_B$$

$$h_R : R_A \leftrightarrow R_B \quad (\text{assumed to be identity})$$

$$h_C : C_B \rightarrow C_A$$

$$\forall(x : D_A) (I_A(x) \Rightarrow I_B(h_D(x)))$$

$$\forall(x : D_A, z : R) (O_A(x, z) \Leftrightarrow O_B(h_D(x), z))$$

$$\forall(x : D_A, z : R, c : C_B) (N\_info_A(x, z, h_C(c)) \Leftrightarrow N\_info_B(h_D(x), z, c))$$

$$\forall(x : D_A, z : R, c : C_B, z' : R)$$

$$(N\_is_A(x, z, h_C(c), z') \Leftrightarrow N\_is_B(h_D(x), z, c, z'))$$

$$\forall(x : D_A, z : R, z' : R) (N_A(x, z, z') \Leftrightarrow N_B(h_D(x), z, z'))$$

$$\forall(x : D_A, z : R, z' : R) (N_A^*(x, z, z') \Leftrightarrow N_B^*(h_D(x), z, z'))$$

Inclusion of the symmetry axiom in the polarity analysis has the effect of making the polarities of N , N_info and N_is neutral if they are not already. The uniqueness axiom assigns a negative polarity to N_info , N_is and C , possibly affecting their overall polarity. As stated earlier, the polarities of C , N_is and N_info are not critical since identity for the first and equivalence for the others can be used. The connection conditions for N and N^* must be verified in light of their definitions. The connection condition for O can safely be modified as described in Section 5.2.4 to include I , either

$$\forall(x : D_A, z : R) (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(h_D(x), z))$$

for reachability or

$$\forall(x : D_A, z : R_A) (I_A(x) \wedge O_A(x, z) \Leftrightarrow O_B(h_D(x), h_R(z)))$$

for feasibility and perfection. Even with this modification, the condition generated by the feasibility axiom, and hence the perfection axiom, is unfortunate—a library neighborhood specification would have to provide an exact match for the output condition of a problem specification for it to guarantee a feasible or perfect neighborhood, and no library is going to have a perfect match for every possible problem. The rest of this section is about dealing with this situation.

6.2.2 Lowry's Design Tactic for Local Search. Lowry's algorithm theory for local search was presented in Chapter I and his tactic very briefly described. This section analyzes his tactic in more detail, in preparation for presenting a revised tactic in the following one. The presentation is adapted to this purpose; it does not match (50) or (51). To assist in analyzing the tactic with respect to the theory developed above and in previous chapters, and since Lowry's design tactic for local search is based on an algebraic theory with sorts and operations similar to those in Figure 68, it will be presented using those names plus symbols from *WFSS* rather than using Lowry's own theory, which was shown in Figure 5. Lowry's work predates Smith's on connections, but is presented so as to facilitate a comparison. On the other hand, Lowry implemented his tactic in KIDS, so some issues that arise in SPECWARE, such as how to combine specifications and identify the appropriate sharing, are not addressed. A summary of his tactic is as follows.

Given a problem spec $Problem \Rightarrow A$, design a local search algorithm for it by carrying out these steps:

1. Retrieve from a library all neighborhood theories with the same output type as the problem and let the user select one; call it B .
2. Derive an input conversion operation, $h_D : D_A \rightarrow D_B$, satisfying

$$\forall(x : D_A, z : R_A) (I_A(x) \wedge O_A(x, z) \Rightarrow I_B(h_D(x)) \wedge O_B(h_D(x), z))$$

3. Complete the interpretation from local search theory to A by defining

$$C_A = C_B$$

$$N\_info_A = \lambda(x : D, z : R, c : C) N\_info_B(h_D(x), z, c)$$

$$N\_is_A = \lambda(x : D, z : R, c : C, z' : C) N\_is_B(h_D(x), z, c, z')$$

4. Derive a *feasibility constraint*, $FC : D, R, C \rightarrow Boolean$, satisfying

$$\forall(x : D, z : R, c : C, z' : R)$$

$$(I(x) \wedge O(x, z) \wedge N\_info(x, z, c) \wedge N\_is(x, z, c, z'))$$

$$\Rightarrow (O(x, z') \Rightarrow FC(x, z, c)))$$

5. Derive an *optimality constraint*, $OC : D, R, C \rightarrow Boolean$, satisfying

$$\forall(x : D, z : R, c : C, z' : R)$$

$$(I(x) \wedge O(x, z) \wedge N\_info(x, z, c) \wedge N\_is(x, z, c, z') \wedge O(x, z'))$$

$$\Rightarrow (Cost(x, z) \leq Cost(x, z') \Rightarrow OC(x, z, c)))$$

6. Instantiate a program schema for hill-climbing, shown in Figure 72.

The condition proved in step 2 is almost but not quite the connection conditions derived for the reachability axiom: it looks like the conditions for I and O combined into one formula, but unfortunately in a way that is strictly weaker and hence does not imply that the correct conditions hold. In practice the formula used is close enough that invalid results seem unlikely to occur. Lowry states that his goal in using this formula was to insure reachability, because all of the neighborhood specifications in the library are reachable.

Lowry provides a library of only four neighborhood specifications. All of them are perfect, but he does not use the term. He refers instead to the invariants that the neighborhood maintains, by which he means solution properties that are not affected by the neighborhood relation. He captures these invariants precisely, so the library neighborhoods are feasible as well as reachable, even though Lowry does not include the feasibility axiom in his algorithm theory. Lowry also states that the B chosen in step 1 should be the one with the strongest invariant that still satisfies

```

function  $F(x : D \mid I(x))$ 
  returns  $(z : R \mid O(x, z) \wedge \forall(c, z') (N\_info(x, z, c) \wedge N\_is(x, z, c, z') \wedge O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z'))))$ 
  =  $F_{LS}(x, FS(x))$ 

function  $FS(x : D \mid I(x))$ 
  returns  $(z : R \mid O(x, z))$ 

function  $F_{LS}(x : D, z : R \mid I(x) \wedge O(x, z))$ 
  returns  $(z' : R \mid O(x, z') \wedge \forall(c, z'') (N\_info(x, z', c) \wedge N\_is(x, z', c, z'') \wedge O(x, z'') \Rightarrow Cost(x, z') \leq Cost(x, z''))))$ 
  = if  $\forall(c, z') N\_info(x, z, c) \wedge N\_is(x, z, c, z') \wedge FC(x, z, c) \wedge O(x, z') \Rightarrow (OC(x, z, c) \wedge Cost(x, z) \leq Cost(x, z'))$ 
  then  $z$ 
  else  $F_{LS}(x, arb(\{z' \mid (c, z') N\_info(x, z, c) \wedge N\_is(x, z, c, z') \wedge FC(x, z, c) \wedge O(x, z') \wedge \neg(OC(x, z, c) \wedge Cost(x, z) \leq Cost(x, z'))\}))$ 

```

Figure 72. Lowry's Program Scheme for Hill Climbing

the condition in step 2. This increases the likelihood that B provides a feasible or nearly feasible neighborhood for A .

Having found a reachable neighborhood in step 2, he then attempts to exclude infeasible solutions from it, trying for a perfect neighborhood. The idea is that given a feasible solution z , determine which of its neighbors are also feasible. Rather than constraining solutions, he formulates his feasibility constraint as a condition on transform variables. In step 3 he derives FC as a necessary condition of the feasibility axiom. Three points are worth noting. First, he is trying to establish a property that is not described in his algorithm theory (see Figure 5 in Chapter I). Second, by deriving FC as a necessary condition, the tactic does not guarantee feasibility. The condition *true* always satisfies the formula, so strong consequents are preferred over weak ones. Feasibility might be achieved, but the tactic doesn't check, so the user will not know. Third, using FC can disrupt the reachability property established earlier. In the extreme, there is no guarantee that *any* neighbors of a feasible solution are feasible—feasible solutions may always be surrounded by infeasible neighbors. For example, in the symmetric traveling salesman problem, one way to represent tours is as a set of edges. One of Lowry's neighborhood specifications is for searching

over subsets of a particular size (essentially the k -subset-1-exchange neighborhood shown above). This provides a reachable neighborhood for this problem, but there are no two tours that differ in exactly one pair of edges: it takes an exchange of at least two edges to transform a tour into another tour. Thus the antecedent in the formula used is *false*, making the strongest consequent *false*. This certainly defines a feasible neighborhood, but not a reachable one. Any time FC is not *true*, the possibility exists that using it to filter the neighborhood found in step 2 will violate reachability. Thus after step 3 the neighborhood with FC incorporated might be perfect, reachable only, feasible only, or neither.

The optimality constraint derived in step 4 is similar to the feasibility constraint of step 3. Lowry wishes to restrict the neighborhood relation to solutions with better cost, in anticipation of using a strict hill-climbing program scheme. Thus this step is inappropriate if some other scheme such as tabu search or simulated annealing were to be used. OC also provides the stopping criteria of the search: if all choices of c satisfy OC , then a local optimum has been reached. As in step 3, however, OC is derived as a necessary condition and so does not guarantee optimality. Using OC can violate reachability, but hill climbing by its nature violates reachability, so this is not as big a concern in this step.

At this point, (51) mentions an optional step that derives necessary conditions for local optimal being global optima. In (50) it says that equivalent conditions are derived. In neither place does Lowry specify a use for the derived conditions in the algorithm.

Hill climbing is the only program scheme Lowry provides, and step 4 shows that the tactic is tailored somewhat to this scheme. Three functions are provided. FS returns an initial feasible solution. Only a specification for this function is provided by the scheme. This specification can then be the basis of a separate algorithm design effort, using whatever kind of algorithm is appropriate. F_{LS} is a hill-climbing iterator, taking a feasible solution and returning a locally optimal solution reachable from it. The output condition shown does not specify that the local optimum returned is

reachable from the solution given because neighborhoods are assumed to be reachable, even though we have shown that they may not be. Finally, F ties these two routines together by passing to F_{LS} the solution returned by FS . The program is written at a very high level of abstraction, as it should be. There is no concern for searching a neighborhood efficiently, selecting the best neighbor vs. the first improved neighbor found, and so forth. Such optimizations and refinements are typically made later in design, after the initial algorithm has been established.

The program scheme enforces feasibility and hill climbing more stringently than the feasibility and optimality constraints derived in previous steps do. At each step of the search, both FC and O are checked for each choice of transform variables and corresponding neighboring solution. This guarantees that only feasible neighbors are searched over, even if FC alone is not strong enough. The intent is that during program optimization the O test will simplify away if FC is strong enough, and even if not, FC may provide a cheaper test that filters out some solutions quickly, before O is checked. Either way, feasibility is maintained. This again runs the risk of violating reachability; for the symmetric TSP example, this program scheme always returns the initial feasible solution returned by FS . Similarly, both OC and local optimality are checked at each step and program optimization may eliminate one of these. Thus FC and OC are little more than optimizations done prior to instantiation of the program scheme. The transformations provided by KIDS for program optimization do not include techniques like the change of variables used for FC and OC , so deriving them in the tactic was necessary if it was to be done at all.

Lowry demonstrates his tactic by deriving a variant of the simplex algorithm. After writing a domain theory for linear algebra, he formulates the linear programming problem as returning the set of basic variables as the solution. The neighborhood selected is k -subset-1-exchange. The feasibility constraint maintains non-negativity by a version of the ratio test. The optimality constraint checks for negative reduced costs. Both of these are strong enough for the O and optimality tests to simplify away once the program scheme is instantiated. Applying the finite differencing transformation of

KIDS to the terms that calculate the basis inverse yields something very like a traditional pivot. Since it is well-known that this neighborhood is exact, the resulting algorithm returns globally optimal solutions when they exist. Complications such as degeneracy, cycling and stalling are not addressed. The tactic works for this case but we have seen that it is not sound in general and is specialized to one program scheme.

6.2.3 A Revised Tactic for Designing Specialized Neighborhoods. We wish to devise a new tactic for matching or specializing neighborhood structures to problem specifications that avoids the problems identified above for Lowry's. We wish to find perfect neighborhoods where possible, and to know whether or not this has been achieved. This information can then be used to guide further design choices concerning move strategy, for example. We also wish to apply the algebraic theory of connections described in Chapter V. We again use the notation $A + B$ to indicate the combination of domain theories A and B with sharing explicitly identified, and we again omit the conversion operation h_R by assuming R_A and R_B are the same sort, denoted simply R . This tactic addresses only the classification task of finding a neighborhood for a problem. The procedure for selecting and instantiating a program scheme was described in Chapter IV; two program schemes for basic local search are presented later in this chapter.

If we attempt to apply the connection tactic described in Chapter V to find a perfect neighborhood, we run into a problem: the equivalence condition on O . One approach to achieving equivalence is to prove implication in one direction and then the other. If we derive conversion operations so that

$$\forall(x : D_A, z : R) (I_A(x) \wedge O_B(h_D(x), z) \Rightarrow O_A(x, z))$$

is true, then we require our library of neighborhood structures to contain highly specialized neighborhoods with strong invariants so that one or more of them will imply the output condition of our problem. This formula by itself also corresponds to no property of neighborhoods that we have

identified. If we instead start from the other implication,

$$\forall(x : D_A, z : R) (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(h_D(x), z))$$

then we require our library to contain general-purpose neighborhoods with weak invariants so that one or more of them will be implied by the output condition of our problem. This situation is much more practical. As a side benefit, this direction of implication corresponds to the connection condition for reachability and so has meaning in terms of neighborhood properties.

With reachability established, we can attempt to prove the other direction of implication. If this succeeds, we have a feasible, and hence perfect neighborhood. Most often this will not succeed, however: the library is unlikely to contain an exact match for our problem. We proceed by attempting to modify the neighborhood so that it is feasible and then proving that the modified neighborhood is still reachable. A definition of N is given in the *Neighborhood* spec. We cannot use a different definition in the mediator or the morphisms we are trying to establish will not exist; moreover, we do not want to force N in this way to have properties that it does not possess in general. We are free, however, to modify the definitions of the components of N that are being derived for the particular problem being solved, and in this way induce the properties we desire. For example, we could change the definition of N_info_A to

$$\begin{aligned} N\_info_A &= \lambda(x : D_A, z : R, c : C) \\ &\quad N\_info_B(h_D(x), z, c) \\ &\quad \wedge \exists(z' : R) (N\_is_B(h_D(x), z, c, z') \wedge O_A(x, z')) \end{aligned}$$

This definition directly restricts the neighborhood relation so that all neighbors of feasible solutions are feasible. Using unskolemization or another constructive technique, we might hope to derive an operator that would describe how to compute z' from x , z and c such that the definition is satisfied. A potential problem with this is that we already have a way of computing z' from these other values,

namely *N_is*. *N_is* is formally a relation, but in practice it is usually functional (for example, in *k*-subset-1-exchange there is a unique *new_A* that satisfies *N_is_{k,1}*, given values for *k*, *S*, *A*, *i* and *j*). *N_is* may in fact not be functional, and if so it may be possible to alter its definition to achieve feasibility, but this seems a remote possibility and we choose to ignore it with no regrets. Moreover, the definition above in effect incorporates *N_is* into *N_info*, eliminating whatever benefit was derived from distinguishing the two aspects of *N* in the first place. Thus what we really want is a set of conditions on the choice of the transform variables that will insure *z'* is feasible. To do this we derive a new expression, *FC* (for *feasibility constraint*), that satisfies the following formula

$$\begin{aligned} & \forall(x : D_A, z : R, c : C, z' : R) \\ & (I_A(x) \wedge O_A(x, z) \wedge N\_info_B(h_D(x), z, c) \\ & \wedge N\_is_B(h_D(x), z, c, z') \Rightarrow (FC(x, z, c) \Leftrightarrow O_A(x, z'))) \end{aligned}$$

This formula is the feasibility axiom with *FC* inserted as being equivalent to *O_A*. The antecedent is a set of assumptions that can be used as context for simplifying *O_A* and reformulating it as *FC*, a condition over transform variables. If we now define

$$N\_info_A = \lambda(x : D, z : R, c : C) (N\_info_B(h_D(x), z, c) \wedge FC(x, z, c))$$

we get a feasible neighborhood. Note that *FC* has the same signature as *N_info*, a relation on transform variables, rather than a function that computes a solution. Also note that, in contrast to Lowry, *FC* is derived as equivalent to *O_A* (in a certain context), not as a consequence of it.

A nice property of *FC* is that it provides a way to tell whether the new solution will be feasible without actually forming it and without evaluating *O_A* directly. This can greatly enhance the efficiency of the resulting program. This effect is secondary, however, to the main purpose of *FC*, which is to ensure feasibility.

Whenever we modify *N_info*, we need to verify that reachability still holds. This proof can be difficult for current automated theorem proving technology, since N^* is in effect a second-order predicate (representing an infinite family of relations, $N_0, N_1, \dots$) and hence requires an inductive proof or something equivalent. Moreover, N is not required (or desired) to be a well-founded order, making induction more difficult. On the other hand, if the original reachability proof for $Neighborhood \Rightarrow B$ is available, it might help guide a new proof. In addition, there are some cases where it is easy to judge the effect of *FC*. If *FC* is *true*, then it does not restrict *N_info* at all and the original library neighborhood is perfect for this problem. On the other hand, if *FC* is *false*, then enforcing feasibility on this library neighborhood will completely disconnect the solution space and clearly should not be used to restrict *N_info*. While there are techniques for searching infeasible neighborhoods (49), this may also suggest that this neighborhood is not well-suited to this problem.

If *FC* is neither *true* nor *false* then at least some feasible solutions have some feasible neighbors, in which case reachability ought to be reverified. A fall-back position is to accept that incorporating *FC* might violate reachability but to prove that each feasible solution has at least one feasible neighbor. The theorem that supports this approach is

$$\begin{aligned}
& \forall (x : D_A, z : R) \\
& \quad (I_A(x) \wedge O_A(x, z) \\
& \quad \Rightarrow \exists (c : C, z' : R) (N\_info_B(h_D(x), z, c) \\
& \quad \quad \wedge N\_is_B(h_D(x), z, c, z') \wedge O_A(x, z')))
\end{aligned}$$

This theorem should be easier to prove than reachability and provides some useful information. This time we do not need a witness for z' ; we simply want to know if the theorem holds. If it does not, then the neighborhood with *FC* is definitely not reachable and is probably not a good choice. If it does, then while there is no guarantee that from a given solution all solutions are reachable,

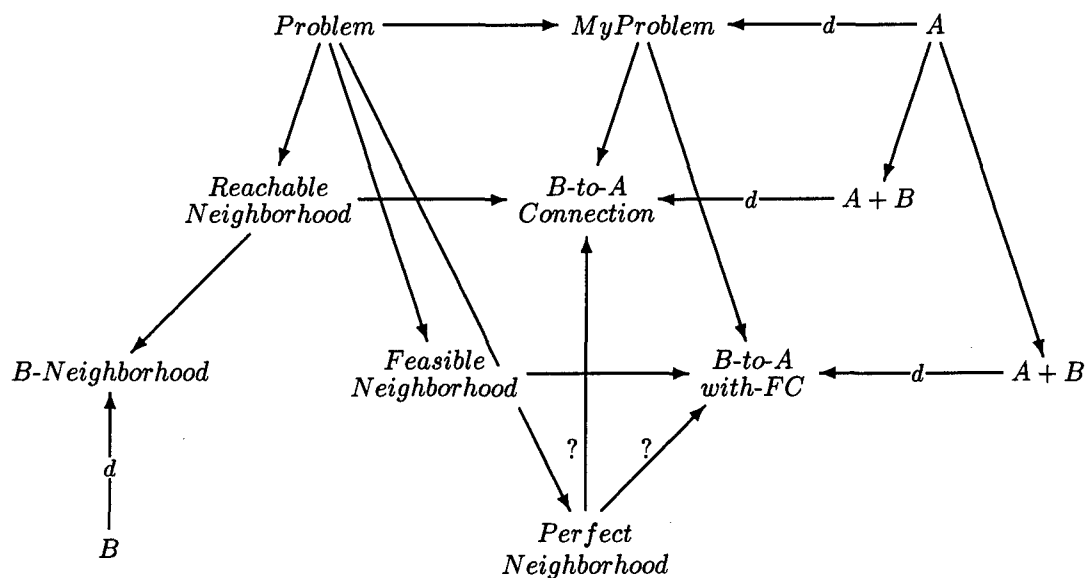


Figure 73. Overview of Neighborhood Matching Tactic

we know that some are. Since the underlying library neighborhood is known to be reachable, one can hope that a restriction that fails to disconnect it totally might leave it reachable.

Putting all the pieces together, the following tactic specializes a library neighborhood to a problem specification using the connection mechanism with elaborations. Since the final result may be less than a perfect neighborhood, the tactic proceeds in two main steps. In the first, a connection is established with respect to *ReachableNeighborhood*, which is the neighborhood spec with only the reachability axiom. In the second, we modify this interpretation to form one from *FeasibleNeighborhood*. We then attempt to establish whether the second one is still reachable and hence a perfect neighborhood. Figure 73 illustrates the process with a diagram. The two interpretation morphisms built are distinct, and even though one is constructed from the other, it does not refine it except in the special case when the two are really the same, in which case they are both perfect. The final result is a specialized neighborhood that is either feasible, reachable or perfect. Most importantly in contrast to Lowry's tactic, upon completion the user *knows* which properties hold and which do not. The tactic is as follows:

1. Given a problem specification $Problem \Rightarrow A$, select from the library a neighborhood structure $ReachableNeighborhood \Rightarrow B$. If a perfect neighborhood for A is sought, a perfect neighborhood should probably be selected as B .
2. Form an initial interpretation scheme $ReachableNeighborhood \Rightarrow A$ via pushout. Rename symbols to follow naming conventions if desired.
3. Combine mediators and target specs of the two interpretations of $ReachableNeighborhood$ by identifying their shared parts and computing colimits. Since reachability requires R_A and R_B to be isomorphic, we assume that these two sorts are included in the shared part and renamed R ; no conversion operation is needed.
4. Add conversion operation

op $h_D : D_A \rightarrow D_B$

connection conditions

axiom $\forall(x : D_A) (I_A(x) \Rightarrow I_B(h_D(x)))$

axiom $\forall(x : D_A, z : R) (I_A(x) \wedge O_A(x, z) \Rightarrow O_B(h_D(x), z))$

and definitions

sort-axiom $C_A = C_B$

definition of N_info_A **is**

axiom $\forall(x : D_A, z : R, c : C) (N\_info_A(x, z, c) \Leftrightarrow N\_info_B(h_D(x), z, c))$

end-definition

definition of N_is_A **is**

axiom $\forall(x : D_A, z : R, c : C, z' : R) (N\_is_A(x, z, c, z') \Leftrightarrow N\_is_B(h_D(x), z, c, z'))$

end-definition

to form the connection spec. Note that the definitions for N_info_A and N_is_A imply that the connection conditions for N and N^* are satisfied, so these are not added.

5. Construct a definition for h_D that satisfies the connection axioms, for example by unskolemization. If successful, this refines $ReachableNeighborhood \Rightarrow A + B$ to an interpretation. If not, abort the tactic.
6. Reformulate O as a feasibility constraint, FC , over transform variables. Copy the mediator from step 5 and add

op $FC : D, R, C \rightarrow Boolean$

axiom $\forall(x : D_A, z : R, c : C, z' : R) (I_A(x) \wedge O_A(x, z) \wedge N\_info_B(h_D(x), z, c) \wedge N\_is_B(h_D(x), z, c, z') \Rightarrow (FC(x, z, c) \Leftrightarrow O_A(x, z'))))$

and derive a definition for FC such that this axiom becomes a theorem. Change the definition of N_info_A to

definition of N_info_A **is**

axiom $\forall(x : D_A, z : R, c : C)$

$(N\_info_A(x, z, c) \Leftrightarrow N\_info_B(h_D(x), z, c) \wedge FC(x, z, c))$

end-definition

This new spec is the mediator of an interpretation $FeasibleNeighborhood \Rightarrow A + B$, but may not be reachable.

7. If $FC = true$ then $O_A = O_B$ and a perfect neighborhood has been found: a morphism exists from $PerfectNeighborhood$ to the mediator derived in step 5. Algorithm design can proceed without worrying about infeasible solutions or unreachability.
8. If $FC = false$ then the solution space of B includes solutions that are infeasible with respect to A and in such a way that no two solutions of A are adjacent. Thus it is not possible to filter out the infeasible solutions and still have a reachable space: the space of feasible solutions with respect to this neighborhood is totally disconnected. Options are to choose a different neighborhood or use the reachable neighborhood derived in step 5 and consider heuristics

that handle infeasible solutions during search, or other techniques such as heuristic repair, relaxation, and so forth.

9. If FC is neither *true* nor *false*, we must decide if FC should be used or not. If not, a reachable neighborhood is still at hand and we have all the options outlined in the previous step. If we choose to use FC , all we have is an interpretation from *FeasibleNeighborhood*. If the neighborhood can be proved exact (see step 10), reachability is not a concern. Otherwise there are two ways to enhance our confidence in this neighborhood.

- (a) Bite the bullet and attempt to prove reachability. If successful, the neighborhood with FC has a morphism from *PerfectNeighborhood*.
- (b) Adopt the “fallback” position described above by proving the following:

$$\begin{aligned} \text{theorem } \forall(x : D_A, z : R) (I_A(x) \wedge O_A(x, z) \\ \Rightarrow \exists(c : C, z' : R) (N\_info_B(h_D(x), z, c) \\ \wedge N\_is_B(h_D(x), z, c, z') \wedge O_A(x, z')))) \end{aligned}$$

This doesn't prove reachability, but it increases our confidence that searching over this neighborhood can produce good results. If this theorem fails to hold, we might reconsider our decision to keep FC .

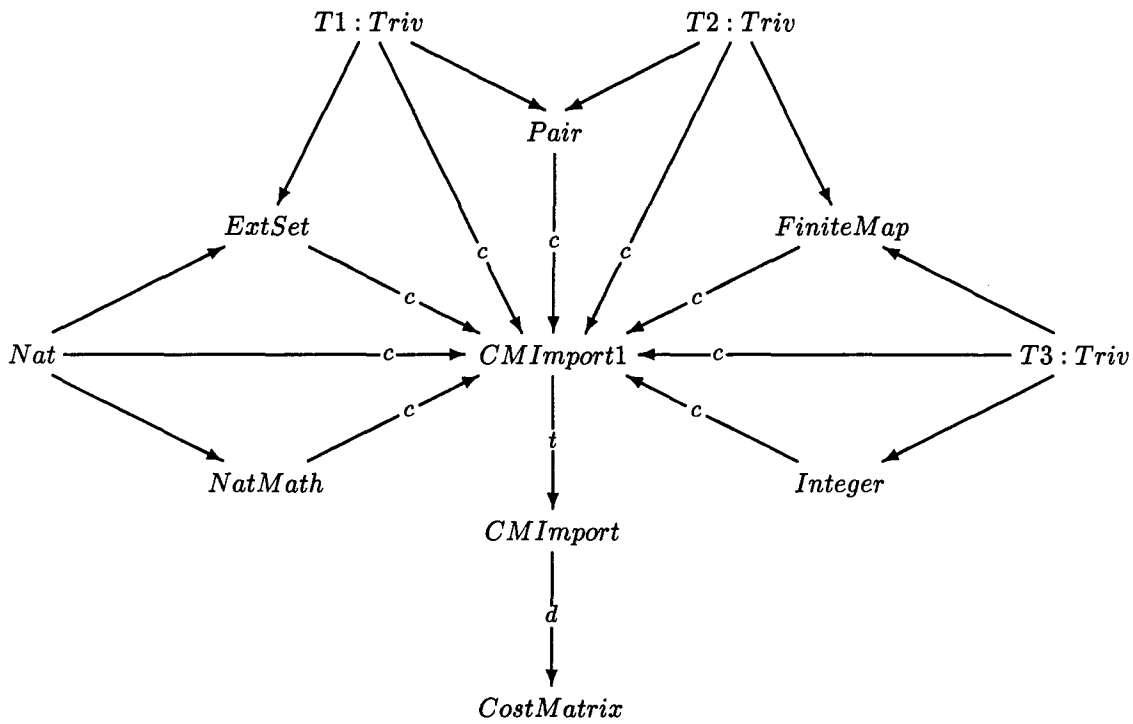
10. Additional neighborhood properties can be proved if desired by constructing morphisms from the specs defining those properties. Symmetry and exactness are the only properties formalized above that have not yet been addressed by the tactic. To prove exactness, one must first extend the interpretation constructed earlier in the tactic by the definitional extension to *LocalOptimum*; this process was explained in Chapter V.

6.2.4 Example. The graph partitioning problem, defined in Chapter III, will provide an example of the neighborhood matching tactic. Graphs have many representations in the mathematical and computer literature. The traditional definition is that a graph $G = (V, E)$ is a set V

of *vertices* and a set E of *edges*, where an edge is a pair of vertices. To be a legal graph, V must be non-empty and E must be a subset of $V \times V$. It would be tedious and uninformative to develop Slang specs for this representation unless one wished to do more extensive work in graph theory. A representation widely used for computer implementations of graph algorithms is the *adjacency matrix*. This is a square, binary matrix where entry a_{ij} is 1 if edge (i, j) is present and 0 otherwise. The graph partitioning problem associates a cost with each edge, so it is convenient to make use of a *cost matrix* where entry c_{ij} is the cost of edge (i, j) and non-existent edges have 0 cost. Rather than develop specs for linear algebra, we can represent such a matrix as a finite map and leverage the specs that have already been presented.

Figures 74 and 75 show how a domain theory for graph partitioning can be built up. The spec *FiniteMap* is the basis of the cost matrix sort. The domain sort of the map is a pair of vertices and the codomain sort is, for convenience, *Integer*. For a map to be used as a cost matrix, it must be *square*. A map is square if it is defined over all pairs from some set of vertices: the domain of the map, considered as a relation on vertices, must be both symmetric and transitive. Another useful property is symmetry. A cost matrix is *symmetric* if it always assigns the equal values to (i, j) and (j, i) . The domain theory includes a sort and operations for sets of vertices and an operation for computing from a cost matrix what the vertices of the graph are. The spec *NatMath*, not shown, extends *Nat* with several standard mathematical operations, including *div* (integer division) and *mod*.

Figure 76 shows the problem specification for graph partitioning in canonical form for an optimization problem, as an interpretation of *WFSS*. The input sort is *CostMatrix* and a valid input must be square and symmetric and its associated vertex set must be non-empty and of size divisible by two. A partition is represented by a set of vertices that is a subset of the vertex set of the input matrix and half its size. The cost function sums the weights of all edges cut by the partition.



spec *Pair* is
 sorts *E, F*
 sort-axiom $F = E, E$
 end-spec

spec *CMImport1* is
 colimit of diagram
 nodes *T1 : Triv, Pair, FiniteMap, T2 : Triv, Integer,*
T3 : Triv, ExtSet, Nat, NatMath
 arcs $T1 \rightarrow Pair : \{\}, T1 \rightarrow ExtSet : \{\},$
 $T2 \rightarrow Pair : \{E \rightarrow F\}, T2 \rightarrow FiniteMap : \{E \rightarrow Dom\},$
 $Nat \rightarrow ExtSet : \{\}, Nat \rightarrow NatMath : \{\},$
 $T3 \rightarrow FiniteMap : \{E \rightarrow Cod\}, T3 \rightarrow Integer : \{E \rightarrow Integer\}$
 end-diagram

spec *CMImport* is
 translate *CMImport* by
 $\{Map \rightarrow CostMatrix, Dom \rightarrow Edge, Cod \rightarrow Integer, T1.E \rightarrow Vertex, Set \rightarrow VSet,$
 $NE-Set \rightarrow NE-VSet, delete \rightarrow v-delete, empty-set \rightarrow empty-vset, in \rightarrow v-in,$
 $insert \rightarrow v-insert, nonempty-set? \rightarrow nonempty-vset?, singleton \rightarrow v-singleton,$
 $union \rightarrow v-union, size \rightarrow v-size, subseteq \rightarrow v-subseteq\}$

Figure 74. Domain Theory for Graph Partitioning

```

spec CostMatrix is
  import CMImport

  op Vertices : CostMatrix → VSet
  definition of Vertices is
    axiom Vertices(empty-map) = empty-vset
    axiom  $\forall (CM : \textit{CostMatrix}, i : \textit{Vertex}, j : \textit{Vertex}, w : \textit{Integer})$ 
      (Vertices(map-shadow(CM,  $\langle i, j \rangle$ , w)) = insert(i, insert(j, Vertices(CM))))
  end-definition

  op square : CostMatrix → Boolean
  definition of square is
    axiom  $\forall (CM : \textit{CostMatrix})$  (square(CM)  $\Leftrightarrow$ 
       $\forall (i : \textit{Vertex}, j : \textit{Vertex})$  (defined-at?(CM,  $\langle i, j \rangle$ )  $\Rightarrow$  defined-at?(CM,  $\langle j, i \rangle$ ))
       $\wedge \forall (i : \textit{Vertex}, j : \textit{Vertex}, k : \textit{Vertex})$ 
        (defined-at?(CM,  $\langle i, j \rangle$ )  $\wedge$  defined-at?(CM,  $\langle j, k \rangle$ )
           $\Rightarrow$  defined-at?(CM,  $\langle i, k \rangle$ )))
  end-definition

  op symmetric : CostMatrix → Boolean
  definition of symmetric is
    axiom  $\forall (CM : \textit{CostMatrix})$  (symmetric(CM)  $\Leftrightarrow$ 
       $\forall (i : \textit{Vertex}, j : \textit{Vertex}, CM\text{-at-}ij : (\textit{CM}, \textit{Edge}) \mid \textit{defined-at?},$ 
        CM-at-ji : (CM, Edge)  $\mid$  defined-at?)
      ( $\langle \textit{CM}, \langle i, j \rangle \rangle = (\textit{relax defined-at?})(\textit{CM-at-}ij)$ 
         $\wedge \langle \textit{CM}, \langle j, i \rangle \rangle = (\textit{relax defined-at?})(\textit{CM-at-}ji)$ 
           $\Rightarrow \textit{map-apply}(\textit{CM-at-}ij) = \textit{map-apply}(\textit{CM-at-}ji))$ )
  end-definition
end-spec

```

Figure 75. Domain Theory for Graph Partitioning, Cont.

$$WFSS \longrightarrow GraphPartition \longleftarrow d \longleftarrow CostMatrix$$

```

spec GraphPartition is
  import CostMatrix

  op GP-I : CostMatrix → Boolean
  definition of GP-I is
    axiom  $\forall (CM : CostMatrix) (GP-I(CM) \Leftrightarrow square(CM) \wedge symmetric(CM) \wedge one \leq size(Vertices(CM)) \wedge size(Vertices(CM)) \bmod two = zero)$ 
  end-definition

  op GP-O : CostMatrix, VSet → Boolean
  definition of GP-O is
    axiom  $\forall (CM : CostMatrix, V : VSet) (GP-O(CM, V) \Leftrightarrow subseteq(V, Vertices(CM)) \wedge v-size(V) = v-size(Vertices(CM)) \div two)$ 
  end-definition

  op CutWeightVertex : CostMatrix, VSet, Vertex → Integer
  definition of CutWeightVertex is
    axiom  $\forall (V : VSet, i : Vertex) (CutWeightVertex(empty-map, V, i) = zero)$ 
    axiom  $\forall (CM : CostMatrix, V : VSet, i : Vertex, j : Vertex, k : Vertex, w : Integer) (i \neq j \Rightarrow CutWeightVertex(map-shadow(CM, \langle j, k \rangle, w), V, i) = CutWeightVertex(CM, V, i))$ 
    axiom  $\forall (CM : CostMatrix, V : VSet, i : Vertex, j : Vertex, w : Integer) (v-in(j, V) \Rightarrow CutWeightVertex(map-shadow(CM, \langle i, j \rangle, w), V, i) = CutWeightVertex(CM, V, i))$ 
    axiom  $\forall (CM : CostMatrix, V : VSet, i : Vertex, j : Vertex, w : Integer) (\neg v-in(j, V) \Rightarrow CutWeightVertex(map-shadow(CM, \langle i, j \rangle, w), V, i) = w + CutWeightVertex(CM, V, i))$ 
  end-definition

  op CutWeightSet : CostMatrix, VSet, Vset → Integer
  definition of CutWeightSet is
    axiom  $\forall (CM : CostMatrix, V : Vset) (CutWeightSet(CM, V, empty-vset) = zero)$ 
    axiom  $\forall (CM : CostMatrix, U : VSet, V : VSet) (CutWeightSet(CM, U, insert(i, V)) = CutWeightVertex(CM, U, i) + CutWeightSet(CM, U, V))$ 
  end-definition

  op CutWeight : CostMatrix, VSet → Integer
  definition of CutWeight is
    axiom  $\forall (CM : CostMatrix, V : VSet) (CutWeight(CM, V) = CutWeightSet(CM, V, V))$ 
  end-definition
end-spec

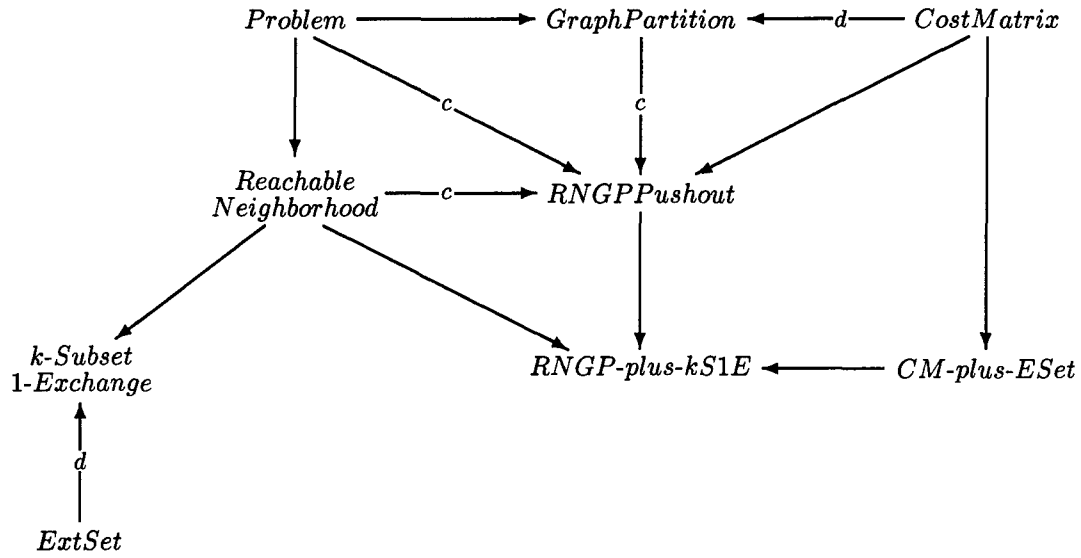
interpretation GraphPartitioning : WFSS  $\Rightarrow$  CostMatrix is
  mediator GraphPartition
  domain-to-mediator
    {D → CostMatrix, I → GP-I, R → VSet, O → GP-O, R → Integer, Cost → CutWeight}
  codomain-to-mediator import-morphism

```

Figure 76. Problem Specification for Graph Partitioning

Figures 77 through 81 show how the neighborhood tactic above is used to define a neighborhood for graph partitioning based on the k -subset-1-exchange neighborhood defined earlier.

1. The problem specification $Problem \Rightarrow CostMatrix$ for the *feasibility problem* of graph partitioning and the neighborhood specification $ReachableNeighborhood \Rightarrow ExtSet$ are arranged as shown in Figure 77. The feasibility problem specification uses the same mediator as the optimization problem specification given above, but uses $Problem$ instead of $WFSS$ as the source spec; the cost domain and cost function are simply ignored. This interpretation shall be referred to as *GPFeasProblem*.
2. An initial interpretation scheme $ReachableNeighborhood \Rightarrow CostMatrix$ is formed by computing a pushout.
3. The interpretation scheme is extended by combining the mediator and target specs of graph partitioning and k -subset-1-exchange. Figure 78 shows the details of the combination. The mediator and target specs are initially combined as coproducts; note how the morphism between them must use qualified names to distinguish the two copies of Nat that each contains. The shared part of $CostMatrix$ and $ExtSet$ is $ExtSet$ itself, so the shape morphism simply adds an arrow between them rather than duplicating $ExtSet$. The colimit of this diagram is isomorphic to $CostMatrix$: the only difference is that many symbols in the colimit have two names, such as Set and $VSet$, or E and $Vertex$. It is simpler to use $CostMatrix$ itself as *CM-plus-ESet* rather than taking the colimit and then having to rename everything again.
4. The mediator spec in step 3 is extended in Figure 79 with a conversion operation h_D , connection conditions for I and O , and definitions for the remaining symbols of *ReachableNeighborhood*.
5. A definition for h_D is derived in Figure 80 from the connection axioms. These axioms and the reachability axiom are now theorems in the mediator spec and are removed. The new spec, *RNGraphPartition*, is therefore a definitional extension of $CostMatrix$. (It is shown

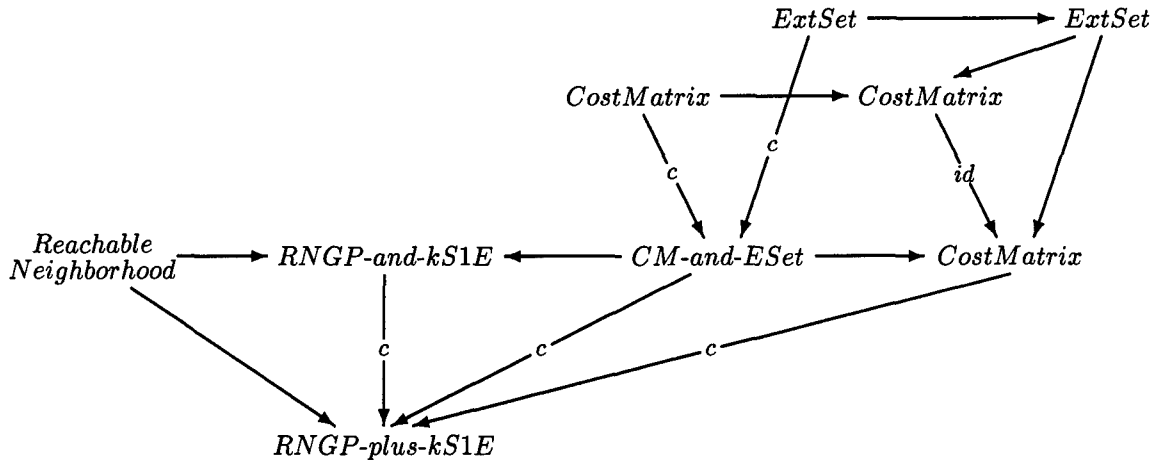


spec *RNGPPushout* is
 colimit of diagram
 nodes *Problem*, *ReachableNeighborhood*, *GraphPartition*
 arcs *Problem* → *ReachableNeighborhood* : **import-morphism**
Problem → *GraphPartition* :
 {*D* → *CostMatrix*, *I* → *GP-I*, *R* → *VSet*, *O* → *GP-O*}
 end-diagram

ip-scheme *RNGP0* : *ReachableNeighborhood* ⇒ *CostMatrix* is
 mediator *RNGPPushout*
 domain-to-mediator cocone-morphism from *ReachableNeighborhood*
 codomain-to-mediator {}

ip-scheme-morphism *GP-to-RNGP0* : *GPFeasProblem* → *RNGP0* is
 domain-sm **import-morphism**
 mediator-sm cocone-morphism from *GraphPartition*
 codomain-sm **identity-morphism**

Figure 77. Neighborhood Tactic for Graph Partitioning, Steps 1–3



spec *RNGP-and-kS1E* is

colimit of diagram nodes *RNGP**Pushout*, *k-Subset-1-Exchange* end-diagram

spec *CM-and-ESet* is colimit of diagram nodes *CostMatrix*, *ExtSet* end-diagram

spec *RNGP-plus-kS1E* is

colimit of diagram

nodes *CM-and-ESet*, *RNGP-and-kS1E*, *CostMatrix*

arcs *CM-and-ESet* $\rightarrow$ *RNGP-and-kS1E* :

{*CostMatrix*.*Nat* $\rightarrow$ *GraphPartition*.*Nat*,
CostMatrix.*zero* $\rightarrow$ *GraphPartition*.*zero*,
CostMatrix.*succ* $\rightarrow$ *GraphPartition*.*succ*,
CostMatrix. $\leq$ $\rightarrow$ *GraphPartition*. $\leq$,
ExtSet.*Nat* $\rightarrow$ *k-Subset-1-Exchange*.*Nat*,
ExtSet.*zero* $\rightarrow$ *k-Subset-1-Exchange*.*zero*,
ExtSet.*succ* $\rightarrow$ *k-Subset-1-Exchange*.*succ*,
ExtSet.*le* $\rightarrow$ *k-Subset-1-Exchange*. $\leq$ },

CM-and-ESet $\rightarrow$ *CostMatrix* :

{*E* $\rightarrow$ *Vertex*, *Set* $\rightarrow$ *VSet*, *NE-Set* $\rightarrow$ *NE-VSet*, *delete* $\rightarrow$ *v-delete*,
empty-set $\rightarrow$ *empty-vset*, *in* $\rightarrow$ *v-in*, *insert* $\rightarrow$ *v-insert*,
nonempty-set? $\rightarrow$ *nonempty-vset?*, *singleton* $\rightarrow$ *v-singleton*,
union $\rightarrow$ *v-union*, *size* $\rightarrow$ *v-size*, *subseq* $\rightarrow$ *v-subseq*}

end-diagram

ip-scheme *RNGP1* : *ReachableNeighborhood* $\Rightarrow$ *CostMatrix* is

mediator *RNGP-plus-kS1E*

domain-to-mediator {*D* $\rightarrow$ *CostMatrix*, *I* $\rightarrow$ *GP-I*, *R* $\rightarrow$ *VSet*, *O* $\rightarrow$ *GP-O*}

codomain-to-mediator cocone-morphism from *CostMatrix*

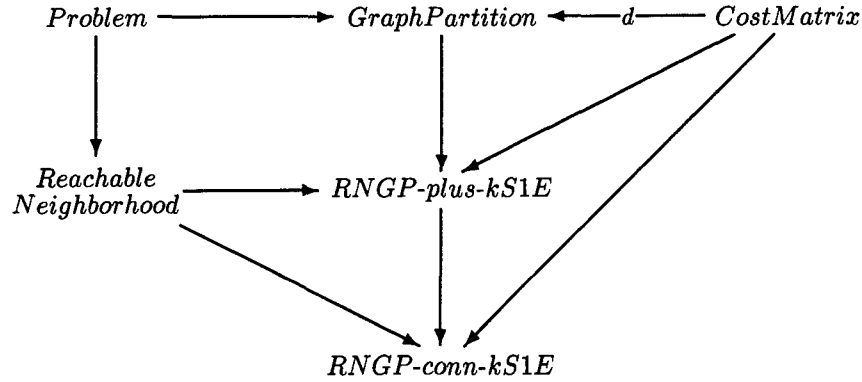
ip-scheme-morphism *GP-to-RNGP1* : *GPFeasProblem* $\rightarrow$ *RNGP1* is

domain-sm import-morphism

mediator-sm {}

codomain-sm identity-morphism

Figure 78. Combining Graph Partitioning and *k*-Subset-1-Exchange



```

spec RNGP-conn-kS1E is
  import RNGP-plus-kS1E
  sort-axiom  $C = KS-C$ 
  op  $h_D : CostMatrix \rightarrow KS-D$ 
  axiom I-Condition is  $\forall (CM : CostMatrix) (GP-I(CM) \Rightarrow KS-I(h_D(CM)))$ 
  axiom O-Condition is
     $\forall (CM : CostMatrix, V : VSet) (GP-O(CM, V) \Rightarrow KS-O(h_D(CM), V))$ 
  definition of N_info is
    axiom  $\forall (CM : CostMatrix, V : VSet, c : C)$ 
       $(N\_info(CM, V, c) \Leftrightarrow Exchange\_info(h_D(CM), V, c))$ 
  end-definition
  definition of N_is is
    axiom  $\forall (CM : CostMatrix, V : VSet, c : C, W : VSet)$ 
       $(N\_is(CM, V, c, W) \Leftrightarrow Exchange(h_D(CM), V, c, W))$ 
  end-definition
end-spec

ip-scheme RNGP2 : ReachableNeighborhood  $\Rightarrow$  CostMatrix is
  mediator RNGP-conn-kS1E
  domain-to-mediator  $\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O\}$ 
  codomain-to-mediator  $\{\}$ 

ip-scheme-morphism GP-to-RNGP2 : GPFeasProblem  $\rightarrow$  RNGP2 is
  domain-sm import-morphism
  mediator-sm  $\{\}$ 
  codomain-sm identity-morphism

```

Figure 79. Neighborhood Tactic for Graph Partitioning, Step 4

importing *RNGP-plus-ESet*, but technically it does not because of the removed axioms; SPECWARE provides no operators for deleting elements from specs, or for transforming specs via unskolemization, for example.) This step shows that *k*-subset-1-exchange can provide a reachable neighborhood for the graph partitioning problem.

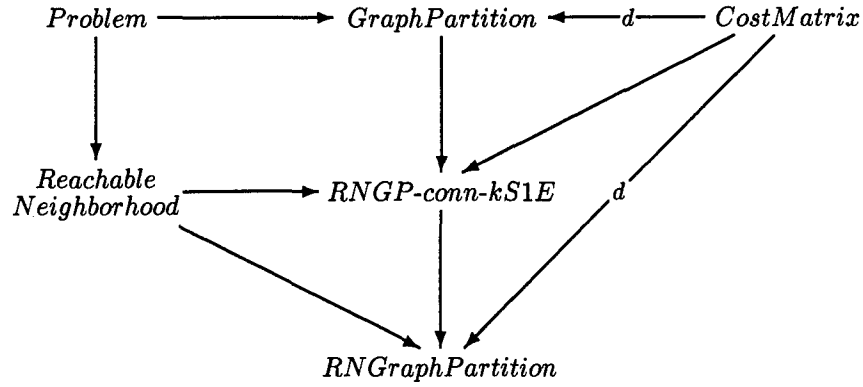
6. The mediator *RNGraphPartition* from the previous step is modified in Figure 81 by adding an operation *FC* and changing the definition of *N_info*, yielding a new spec *FNGraphPartition* (which again does not truly import *RNGP-plus-ESet*—we certainly do not want the reachability axiom now!). A definition is derived for *FC* such that

$$\begin{aligned} & \forall (CM : CostMatrix, V : VSet, c : C, W : VSet) \\ & (GP-I(CM) \wedge GP-O(CM, V) \wedge Exchange\_info(h_D(CM), V, c) \\ & \wedge Exchange(h_D(CM), V, c, W) \Rightarrow (FC(CM, V, c) \Leftrightarrow GP-O(CM, W))) \end{aligned}$$

is a theorem. The definition *FC* = *true* is derived. This modifies *k*-subset-1-exchange to a feasible neighborhood for graph partitioning.

7. Since *FC* = *true*, the feasibility axiom is a theorem in *RNGraphPartition* (and the reachability axiom is a theorem in *FNGraphPartition*); the symmetry axiom is also satisfied (in both). Therefore the interpretation derived in step 5 can be extended to one from *Perfect-SymmetricNeighborhood* without modifying the mediator or target specs.

The final mediator spec can be cleaned up as shown in Figure 82 to get rid of unnecessary levels of definition. The refinements *Problem* $\Rightarrow$ *CostMatrix*, *WFSS* $\Rightarrow$ *CostMatrix* and *Perfect-SymmetricNeighborhood* $\Rightarrow$ *CostMatrix* are consistent over their shared part—how they refine *Problem*—so they can be combined to form an interpretation *PSLocalSearch* $\Rightarrow$ *CostMatrix*, shown in Figure 83. This completes the classification of graph partitioning as a local search problem and prepares it for the second part of algorithm design, that of choosing and instantiating a program scheme.



```

spec RNGraphPartition is
  import RNGP-plus-kS1E

  sort-axiom  $C = KS-C$ 

  op  $h_D : CostMatrix \rightarrow KS-D$ 
  definition of  $h_D$  is
    axiom  $\forall(CM : CostMatrix)$ 
       $(h_D(CM) = (size(Vertices(CM)) \text{ div } two, Vertices(CM)))$ 
  end-definition

  theorem I-Condition is  $\forall(CM : CostMatrix) (GP-I(CM) \Rightarrow KS-I(h_D(CM)))$ 
  theorem O-Condition is
     $\forall(CM : CostMatrix, V : VSet) (GP-O(CM, V) \Rightarrow KS-O(h_D(CM), V))$ 

  definition of N_info is
    axiom  $\forall(CM : CostMatrix, V : VSet, c : C)$ 
       $(N\_info(CM, V, c) \Leftrightarrow Exchange\_info(h_D(CM), V, c))$ 
  end-definition

  definition of N_is is
    axiom  $\forall(CM : CostMatrix, V : VSet, c : C, W : VSet)$ 
       $(N\_is(CM, V, c, W) \Leftrightarrow Exchange(h_D(CM), V, c, W))$ 
  end-definition

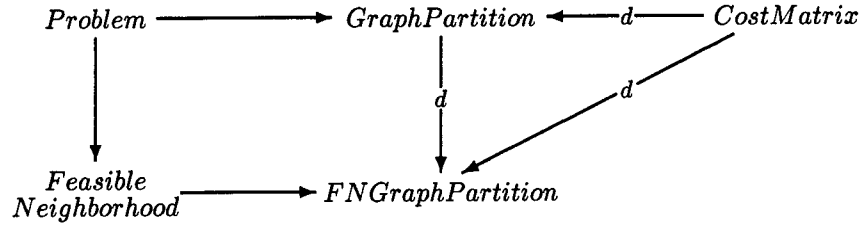
end-spec

interpretation RNGraphPartitioning : ReachableNeighborhood  $\Rightarrow$  CostMatrix is
  mediator RNGraphPartition
  domain-to-mediator  $\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O\}$ 
  codomain-to-mediator  $\{\}$ 

ip-scheme-morphism GP-to-RNGP : GPFeasProblem  $\rightarrow$  RNGraphPartitioning is
  domain-sm import-morphism
  mediator-sm  $\{\}$ 
  codomain-sm identity-morphism

```

Figure 80. Neighborhood Tactic for Graph Partitioning, Step 5



```

spec FNGraphPartition is
  import RNGP-plus-ESet

  sort C
  sort-axiom C = KS-C

  op  $h_D : \text{CostMatrix} \rightarrow \text{KS-D}$ 
  definition of  $h_D$  is
    axiom  $\forall (CM : \text{CostMatrix}) (h_D(CM) = \langle \text{size}(\text{Vertices}(CM)) \text{ div } \text{two}, \text{Vertices}(CM) \rangle)$ 
  end-definition

  theorem I-Condition is  $\forall (CM : \text{CostMatrix}) (GP-I(CM) \Rightarrow KS-I(h_D(CM)))$ 
  theorem O-Condition is
     $\forall (CM : \text{CostMatrix}, V : \text{VSet}) (GP-O(CM, V) \Rightarrow KS-O(h_D(CM), V))$ 

  op  $FC : \text{CostMatrix}, \text{VSet}, C \rightarrow \text{Boolean}$ 
  definition of FC is
    axiom  $\forall (CM : \text{CostMatrix}, V : \text{VSet}, c : C) FC(CM, V, c)$ 
  end-definition
  theorem  $\forall (CM : \text{CostMatrix}, V : \text{VSet}, c : C, W : \text{Vset})$ 
     $(GP-I(CM) \wedge GP-O(CM, V) \wedge \text{Exchange\_info}(h_D(CM), V, c)$ 
       $\wedge \text{Exchange}(h_D(CM), V, c, W) \Rightarrow (FC(CM, V, c) \Leftrightarrow GP-O(CM, W)))$ 

  definition of N_info is
    axiom  $\forall (CM : \text{CostMatrix}, V : \text{VSet}, c : C)$ 
       $(N\_info(CM, V, c) \Leftrightarrow \text{Exchange\_info}(h_D(CM), V, c) \wedge FC(CM, V, c))$ 
  end-definition

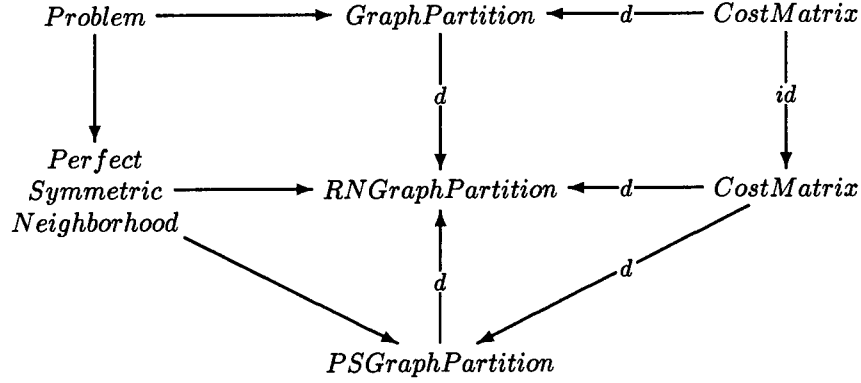
  definition of N_is is
    axiom  $\forall (CM : \text{CostMatrix}, V : \text{VSet}, c : C, W : \text{VSet})$ 
       $(N\_is(CM, V, c, W) \Leftrightarrow \text{Exchange}(h_D(CM), V, c, W))$ 
  end-definition
end-spec

interpretation FNGraphPartitioning : ReachableNeighborhood  $\Rightarrow$  CostMatrix is
  mediator FNGraphPartition
  domain-to-mediator  $\{D \rightarrow \text{CostMatrix}, I \rightarrow GP-I, R \rightarrow \text{VSet}, O \rightarrow GP-O\}$ 
  codomain-to-mediator  $\{\}$ 

ip-scheme-morphism GP-to-FNGP : GPFeasProblem  $\rightarrow$  FNGraphPartitioning is
  domain-sm import-morphism
  mediator-sm  $\{\}$ 
  codomain-sm identity-morphism

```

Figure 81. Neighborhood Tactic for Graph Partitioning, Step 6



```

spec PSGraphPartition is
  import GraphPartition

  sort C
  sort-axiom C = Vertex, Vertex

  op Exchange_info : CostMatrix, VSet, C → Boolean
  definition of Exchange_info is
    axiom  $\forall (CM : CostMatrix, V : VSet, i : Vertex, j : Vertex)$ 
       $(Exchange\_info(CM, V, \{i, j\}) \Leftrightarrow in(i, Vertices(CM)) \wedge \neg in(i, V) \wedge in(j, V))$ 
  end-definition

  op Exchange : CostMatrix, VSet, C, VSet → Boolean
  definition of Exchange is
    axiom  $\forall (CM : CostMatrix, V : VSet, i : Vertex, j : Vertex, W : VSet)$ 
       $(Exchange(CM, V, \{i, j\}, W) \Leftrightarrow W = insert(i, delete(j, V)))$ 
  end-definition

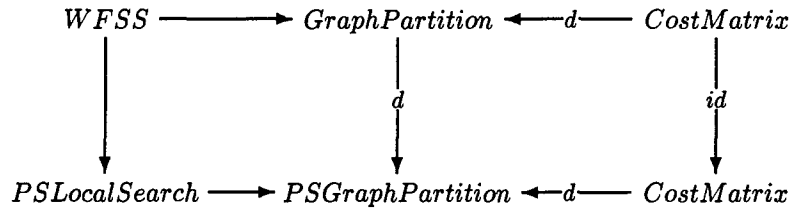
  op N : CostMatrix, VSet, C → Boolean
  definition of N is
    axiom  $\forall (CM : CostMatrix, V : VSet, W : VSet)$ 
       $(N(CM, V, W) \Leftrightarrow \exists (c : C) (Exchange\_info(CM, V, c) \wedge Exchange(CM, V, c, W)))$ 
  end-definition

  op N* : CostMatrix, VSet, C → Boolean
  definition of N* is
    axiom  $\forall (CM : CostMatrix, V : VSet, W : VSet)$ 
       $(N^*(CM, V, W) \Leftrightarrow V = W \vee \exists (U : VSet) (N(CM, V, U) \wedge N^*(CM, U, W)))$ 
  end-definition
end-spec

interpretation PSGraphPartitioning : PerfectSymmetricNeighborhood  $\Rightarrow$  CostMatrix is
  mediator PSGraphPartition
  domain-to-mediator {D → CostMatrix, I → GP-I, R → VSet, O → GP-O,
    N_info → Exchange_info, N.is → Exchange}
  codomain-to-mediator {}

```

Figure 82. Cleaning Up the Neighborhood for Graph Partitioning



interpretation *PSLSGraphPartitioning* : *PSLocalSearch* $\Rightarrow$ *CostMatrix* is
mediator *PSGraphPartition*

domain-to-mediator

$\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O, \mathcal{R} \rightarrow Integer,$
 $Cost \rightarrow CutWeight, N\_info \rightarrow Exchange\_info, N\_is \rightarrow Exchange\}$

codomain-to-mediator $\{\}$

ip-scheme-morphism *GP-to-LSGP* : *GraphPartitioning* $\Rightarrow$ *PSLSGraphPartitioning* is
domain-sm cocone-morphism from *WFSS*

mediator-sm import-morphism

codomain-sm identity-morphism

Figure 83. Local Search Specification for Graph Partitioning

6.3 Program Schemes for Basic Local Search

The elements of local search that are not formalized in *LocalSearch*—overall strategy, finding an initial solution, move selection rules and stopping criteria—are addressed in the program scheme or deferred to later in design. Design decisions are deferred by extending the domain theory for local search. For example, finding an initial solution is largely independent of other design decisions and is a significant subtask in its own right. To support the top-down design of an algorithmic solution to this problem, all program schemes extend the domain theory with some variant of an operator and an axiom

op *InitSol* : $D \rightarrow R$

axiom $\forall(x : D) (I(x) \Rightarrow O(x, InitSol(x)))$

specifying the task of finding a feasible solution, with no further detail.

Other extensions are typical of local search program schemes. At each step of a search certain state information must be maintained. The program scheme must include sorts for this information,

a predicate for characterizing its legal values, a means for initializing it at the beginning of a search, and a means for updating it. The current solution, for example, is part of this state information. The sort R has already been defined, O defines its legal values and $InitSol$ is the initialization operation; it is maintained by choosing a move and applying it. Particular program schemes may need additional state information. Some of the design issues involved will be fully defined in the mediator of the program scheme interpretation. Others will be deferred by placing them in the target spec as extensions to the domain theory.

Another common feature of local search program schemes is the need for inputs in addition to D , the problem instance to solve. Some of these inputs might be tunable parameters that will eventually be fixed after experimentation is used to determine suitable values, either as constants or computed somehow for each problem instance. In the absence of robust methods for setting such parameters, however, the end user may be required to provide values as part of the input data, to allow experimentation in the fielded system. Finally, if randomizing techniques are used, some means for initializing a random number generator will be needed.

The problem with all of these additional inputs is that the input sort was long ago established as D and cannot be changed. Both *Problem* and *Program* include D , and the use of interpretation morphisms require that the D identified in the original problem specification be carried through to the program that solves it, at least in the left-hand square that maps D to the mediators. It is possible, however, to do some things in the mediator and target specs to achieve the desired effect.

The dilemma is resolved by a mechanism similar to that used for state information. Sorts and operations are introduced to represent, initialize and update whatever additional information is required. The "main program" to which F will be mapped will accept an input, call all the initialization operations and then call an auxiliary routine with this expanded set of inputs. The auxiliary operation does all the work, and the modified interface is hidden from the source spec. Some of the sorts and operations needed for this approach may be definable in the mediator, but by

definition some of them will not: if all the answers were known, the inputs would not be needed in the first place. Whatever is not known is added to the domain theory as deferred decisions. Future refinements will define them in terms of simpler domain theories, as required by code generation if not sooner. At some point, presumably, a user interface, or a system clock for seeding a random number generator, or whatever else is needed will be added to the specification and can be used explicitly. This technique is a little clumsy and inconsistent—it is not needed for inputting a problem instance, for example—but something like it is required by the adopted formalism, and it is general enough to do the job.

We said in Chapter IV that the program scheme contained only non-problem-dependent extensions, but now this is no longer true: inputs and state information are very much problem-dependent. This does not affect the instantiation process, however, precisely because the design decisions involved are being deferred to later refinements. During instantiation we explicitly state that the knowledge needed to define these extensions is not yet available. Therefore, identity completion is completely appropriate.

An alternative to deferring decisions in this way is to treat the problem-dependent extensions required by the program scheme as an additional classification step to be done before instantiation. Why would one choose to instantiate a scheme first and then finish the classification later? The primary benefit is to reduce the conceptual load on the designer. By instantiating a scheme, the designer is making a major design decision and committing to it. The details can be worked out at leisure. In practice it does not matter in what order these steps are done; the same result can be achieved either way. Even the neighborhood structure could be worked out after a program scheme has been chosen, if desired. Whichever approach seems easier to understand and to keep track of what remains to be done should be adopted. The steps in local search design have a certain logical sequence, however, that it can be useful to follow. Neighborhood structure is crucial to all schemes; without a neighborhood, there is no local search. Further, some program schemes depend on certain

neighborhood properties. It makes sense to insure that a neighborhood with those properties can be achieved before committing to a scheme. With that done, instantiating a scheme collects all the remaining tasks in one place, in an explicit context where the designer can see what the sorts and operations are used for. These tasks may only need to be done because a particular scheme was chosen, so it makes sense to get the instantiation out of the way before proceeding. Ultimately the ordering of tasks is another issue for the taste and judgment of the designer, who should be allowed to choose for him or herself.

6.3.1 Hill Climbing. The prototypical local search algorithm is hill climbing. Hill climbing computes a local optimum by starting from an initial feasible solution and moving at each step to a neighbor of better cost until a local optimum is reached. Hill climbing requires a feasible neighborhood to guarantee a feasible solution is returned. Multiple searches can be performed from different starting points to get a sample of local optima; the best of these will be returned. To contrast it with hill climbing that allows neutral moves, it is sometimes called *strict*.

Figures 84 through 86 show a program scheme for strict hill climbing. Figure 84 shows how *FLocalSearch* (which is *LocalSearch* with the feasibility axiom; see Figure 69) is extended to a domain theory *HillClimb*. This theory adds several new sorts and operators that will be needed by the algorithm itself, which is given in the mediator. Some of these, like the spec *Nat*, support computational structures such as iteration. Others represent design choices that have not yet been made.

The only added input parameter for this scheme is the number of times to search. The sort for this number is *Pos*, meaning any non-zero natural number. The operation *MaxTries* computes this value. The spec for *Nat* given in Chapter II defines *Pos* as a subsort of *Nat* characterized by the predicate *nonzero?*.

ISP is a sort for *initial solution parameters*. Since multiple searches will be done, multiple initial feasible solutions are required. Initial solution parameters represent data needed to vary

```

spec HillClimb is
  import FLocalSearch, Nat

  sort ISP

  op LegalISP : D, ISP  $\rightarrow$  Boolean

  op InitISP : D  $\rightarrow$  ISP
  axiom  $\forall (x : D) (I(x) \Rightarrow \text{LegalISP}(x, \text{InitISP}(x)))$ 

  op InitSol : D, ISP  $\rightarrow$  R, ISP
  axiom  $\forall (x : D, p : ISP)$ 
     $(I(x) \wedge \text{LegalISP}(x, p) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$ 
  axiom  $\forall (x : D, p : ISP)$ 
     $(I(x) \wedge \text{LegalISP}(x, p) \Rightarrow \text{LegalISP}(x, (\text{project } 2)(\text{InitSol}(x, p))))$ 

  op MaxTries : D  $\rightarrow$  Pos

  op BetterNeighbor : D, R  $\rightarrow$  R
  axiom  $\forall (x : D, z : R)$ 
     $(\exists (z' : R) (N(x, z, z') \wedge \text{Cost}(x, z') < \text{Cost}(x, z))$ 
       $\Rightarrow N(x, z, \text{BetterNeighbor}(x, z))$ 
       $\wedge \text{Cost}(x, \text{BetterNeighbor}(x, z)) < \text{Cost}(x, z))$ 

end-spec

```

Figure 84. Domain Theory for Strict Hill Climbing

the solution generated. It could be a random number generator seed, a modification to the cost function, or data gathered during previous searches, for example. The operations *LegalISP* and *InitISP* define legal and initial values, respectively. An axiom states that the initial value is legal.

The operator *InitSol* computes an initial feasible solution. It takes a problem instance and an initial solution parameter and computes a feasible solution and a new initial solution parameter. The spec does not say whether *InitSol* actually changes the input parameter passed; the only requirement is that the value returned be legal. The output solution is required only to be feasible.

The operator *BetterNeighbor* will be used during search to select a neighbor of the current solution. For hill climbing this solution is required to be strictly better than the current solution. If there is a choice of better solution, the axiom does not state which is selected. If there are no better solutions, the behavior is again undefined.

The mediator *HCPProgram* defines the hill climbing algorithm; it is shown in Figure 85. The operation *HillClimbMain* starts things off by getting initial values for the initial solution

parameters and the number of trials to run, then calls *HillClimbIter* to perform the specified number of searches. For each search, *HillClimbIter* gets an initial solution and calls *HillClimbStep* to climb to a local optimum. *HillClimbIter* returns the local optimum of minimal cost among those computed so far, and also a legal initial solution parameter that is passed to the next search, if any. The local optimum of minimum cost over all searches is returned by *HillClimbMain* as the answer. The scheme does not show *HillClimbStep* using or modifying the initial solution parameter; only *InitSol* can do that. If information gathered during a climb were somehow to be used to influence the choice of initial solution in later searches, more sorts and operations would have to be added.

Figure 86 completes the hill climbing program scheme by showing an interpretation *HillClimbing* and an interpretation morphism to it from *FLocalOptimization*. This morphism can be composed with *LO-to-FLO* to form a morphism from *LocalOptimization* to *HillClimbing*.

The theorems at the end of *HCPProgram* capture the properties of the hill climbing operations that are considered essential. The last one is the image of the correctness axiom in *Program*. This theorem must hold if the morphism from *Program* is to be valid. The other theorems serve as lemmas.

Theorem 6.3.1 *In the spec HCPProgram,*

$$\forall(x : D, z : R) (I(x) \wedge O(x, z) \Rightarrow LO(x, HillClimbStep(x, z)))$$

Proof. Let x and z satisfy $I(x) \wedge O(x, z)$. If z is a local optimum, then *HillClimbStep* returns it and the theorem is satisfied. If not, then a neighbor z' of z with better cost is selected; by definition of local optimality, such a neighbor must exist. Since N is a feasible neighborhood, z' is a feasible solution. This solution is passed recursively to *HillClimbStep*. Since z' is strictly better than z , the recursion terminates with a feasible, locally optimal solution. $\square$

```

spec HCP-Import is
  colimit of diagram
    nodes FLocalSearch, HillClimb, FLocalOptimum
    arcs  FLocalSearch  $\rightarrow$  FLocalOptimum :
          cocone-morphism from FLocalSearch
          FLocalSearch  $\rightarrow$  HillClimb : {}
    end-diagram

spec HCPProgram is
  import HCP-Import

  op BetterSol : D, R, R  $\rightarrow$  R
  definition of BetterSol is
    axiom  $\forall (x : D, z1 : R, z2 : R) (Cost(x, z1) \leq Cost(x, z2) \Rightarrow BetterSol(x, z1, z2) = z1)$ 
    axiom  $\forall (x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow BetterSol(x, z1, z2) = z2)$ 
  end-definition

  op HillClimbMain : D  $\rightarrow$  R
  definition of HillClimbMain is
    axiom  $\forall (x : D) (HillClimbMain(x) = HillClimbIter(x, MaxTries(x), InitISP(x)))$ 
  end-definition

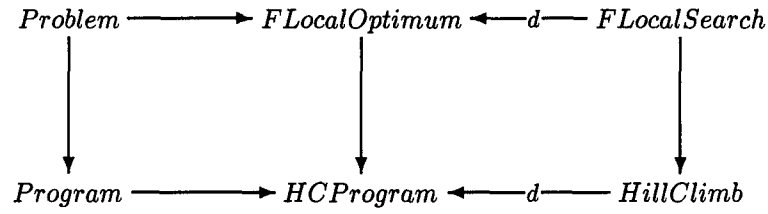
  op HillClimbIter : D, Pos, ISP  $\rightarrow$  R
  definition of HillClimbIter is
    axiom  $\forall (x : D, p : ISP, z : R)$ 
       $(z = (\text{project } 1)(InitSol(x, p)) \Rightarrow HillClimbIter(x, p, succ(zero)) = HillClimbStep(x, z))$ 
    axiom  $\forall (x : D, k : Pos, p : ISP, z : R, p' : ISP, z' : R, z'' : R)$ 
       $((z, p') = InitSol(x, p) \wedge z' = HillClimbStep(x, z) \wedge z'' = HillClimbIter(x, k, p'))$ 
       $\Rightarrow HillClimbIter(x, succ(nat-of-pos(k)), p) = BetterSol(x, z', z'')$ 
  end-definition

  op HillClimbStep : D, R  $\rightarrow$  R
  definition of HillClimbStep is
    axiom  $\forall (x : D, z : R) (LocallyOptimal(x, z) \Rightarrow HillClimbStep(x, z) = z)$ 
    axiom  $\forall (x : D, z : R)$ 
       $(\neg LocallyOptimal(x, z) \Rightarrow HillClimbStep(x, z) = HillClimbStep(x, BetterNeighbor(x, z)))$ 
  end-definition

  theorem  $\forall (x : D, z : R) (I(x) \wedge O(x, z) \Rightarrow LO(x, HillClimbStep(x, z)))$ 
  theorem  $\forall (x : D, k : Pos, p : ISP)$ 
     $(I(x) \wedge LegalISP(x, p) \Rightarrow LO(x, HillClimbIter(x, k, p)))$ 
  theorem  $\forall (x : D) (I(x) \Rightarrow LO(x, HillClimbMain(x)))$ 
end-spec

```

Figure 85. Mediator Spec for Strict Hill Climbing



interpretation *HillClimbing* : *Program* $\Rightarrow$ *HillClimb* is
mediator *HCProgram*
domain-to-mediator $\{O \rightarrow LO, F \rightarrow HillClimbMain\}$
codomain-to-mediator $\{\}$

ip-scheme-morphism *FLO-as-HillClimbing* : *FLocalOptimization* $\rightarrow$ *HillClimbing* is
domain-sm import-morphism
mediator-sm $\{\}$
codomain-sm $\{\}$

Figure 86. Program Scheme for Local Optimization via Strict Hill Climbing

Theorem 6.3.2 *In the spec HCProgram,*

$$\forall(x : D, k : Pos, p : ISP)$$

$$(I(x) \wedge LegalISP(x, p) \Rightarrow LO(x, HillClimbIter(x, k, p)))$$

Proof. Let x and p satisfy $I(x) \wedge LegalISP(x, p)$ and let k be a positive number. Assume $k = succ(zero)$ (a.k.a one). *HillClimbIter* will call *InitSol*, and since x is a valid input, an axiom for *InitSol* guarantees that it will return a feasible solution, z . By Theorem 6.3.1, *HillClimbStep* will then compute a feasible, locally optimal solution, z' . This solution is returned unchanged by *HillClimbIter*.

Now assume the theorem holds for $one < k < n$ and let $k = n$. *HillClimbIter* will call *InitSol* with a valid input and get a feasible solution, z , and a valid initial solution parameter, p' . It will pass z to *HillClimbStep* and by the first theorem get back a feasible, locally optimal solution, z' . It will pass p' , along with x and $k - 1$, recursively to *HillClimbIter*, and by the

induction hypothesis will get back another feasible, locally optimal solution, z'' . *BetterSol* then selects either z' or z'' as the one returned. $\square$

Theorem 6.3.3 *In the spec HCPProgram,*

$$\forall(x : D) (I(x) \Rightarrow LO(x, HillClimbMain(x)))$$

Proof. Let x satisfy $I(x)$. The axioms for *MaxTries* and *InitISP* guarantee that they will return valid values for k and p , respectively. These are passed to *HillClimbIter*, which by the previous theorem produces a feasible, locally optimal solution. $\square$

6.3.2 Hill Climbing with a Single Trial. There are cases, such as search over an exact neighborhood, when one search is sufficient. One could provide a separate program scheme for one-trial hill climbing, but logically it is a minor refinement of the existing scheme and it can be constructed as such. Figure 87 shows a refinement *HillClimbOnce*,

$$HillClimb \longrightarrow HillClimbOnce \longleftarrow^d HillClimb1$$

that greatly simplifies the target spec. The new target, *HillClimb1*, adds only the spec *Nat* and the operation *InitSol* to *FLocalSearch*. *Nat* is still needed for defining *MaxTries* in the mediator. The operation *BetterNeighbor* is unchanged. The sort *ISP* and its operations have been done away with and *InitSol* has a reduced signature: it takes an input and produces a solution. Valid inputs yield feasible solutions, as before. The mediator uses some interesting techniques to define the components of *HillClimb* in terms of those of *HillClimb1*. The sort *ISP* that was eliminated in the target is defined here as the empty product sort, denoted $\langle \rangle$. This sort is built in to every spec. It has a single element, denoted by the empty tuple, $\langle \rangle^1$. The operation *LegalISP* says that $\langle \rangle$ is a legal parameter, and *InitISP* returns it. An operation *InitSol1* is defined that ignores the

¹In the topos that a spec generates, the empty product sort is a *terminal object*: for every object (sort) there is a unique arrow (operation) from it to this object

initial solution parameter and just calls *InitSol*. Finally, the operation *MaxTries* is defined to ignore its input and always return one.

This refinement can be composed with the program scheme *HillClimbing* to give a new scheme, *HillClimbOnce*, for single-trial hill climbing. This construction is shown in Figure 88. This scheme does what we want, but the definitions for *HillClimbMain*, *HillClimbIter* and *HillClimbStep* in the mediator, *HCMaXOnceProgram*, are the same as they were before—they do not take advantage of the specialization of *MaxTries* to one. Figure 89 shows a simpler mediator, *HC1Program*. *HillClimbMain* has been changed to call *HillClimbStep* directly, eliminating *HillClimbIter* completely. All references to *ISP* are gone. *Nat* is still present, but is never used. This program scheme also solves the local optimization problem, as the interpretation morphism from *FLocalOptimization* shows. More importantly, however, there is an interpretation morphism from *HillClimbing1* to *HillClimbingOnce*, one composed entirely of definitional extensions. Figure 90 shows these relationships. The shape of this diagram is that of a classification diagram. One difference in its construction is that the parts that were given were the top and right interpretations, rather than top and left. The middle interpretation is derived uniquely via colimit and the bottom was simplified by hand to take advantage of the simpler structure of *HillClimb1*. If desired, an additional simplification step could be carried out to remove *Nat* from *HillClimb1* and *HC1Program*, yielding a new program scheme with an interpretation morphism to *HillClimbing1*. The components of this morphism would not be definitional extensions, however.

6.3.3 Steepest Ascent Hill Climbing. A more substantial refinement of hill climbing focuses on how *HillClimbStep* chooses a move. In the scheme above, any improving move will do. If this choice is specialized to a most improving move, or best neighbor, then the result is a program scheme for *steepest ascent* hill climbing.

Figure 91 shows the refinement. Spec *SAHillClimb* is the new target theory. It does not import *HillClimb*, but is identical to it except that the new operation *BestNeighbor* replaces

```

spec HillClimb1 is
  import FLocalSearch, Nat

  op InitSol :  $D \rightarrow R$ 
  axiom  $\forall(x : D) (I(x) \Rightarrow O(x, \text{InitSol}(x)))$ 

  op BetterNeighbor :  $D, R \rightarrow R$ 
  axiom  $\forall(x : D, z : R)$ 
     $(\exists(z' : R) (N(x, z, z') \wedge \text{Cost}(x, z') < \text{Cost}(x, z))$ 
       $\Rightarrow N(x, z, \text{BetterNeighbor}(x, z))$ 
       $\wedge \text{Cost}(x, \text{BetterNeighbor}(x, z)) < \text{Cost}(x, z))$ 

end-spec

spec HillClimbOnce is
  import HillClimb1

  sort ISP
  sort-axiom ISP = ()

  op LegalISP :  $D, ISP \rightarrow \text{Boolean}$ 
  definition of LegalISP is
    axiom  $\forall(x : D, p : ISP) \text{LegalISP}(x, p)$ 
  end-definition

  op InitISP :  $D \rightarrow ISP$ 
  definition of InitISP is
    axiom  $\forall(x : D) (\text{InitISP}(x) = \langle \rangle)$ 
  end-definition

  op InitSol1 :  $D, ISP \rightarrow \text{Boolean}$ 
  definition of InitSol1 is
    axiom  $\forall(x : D, p : ISP) (\text{InitSol1}(x, p) = \text{InitSol}(x))$ 
  end-definition

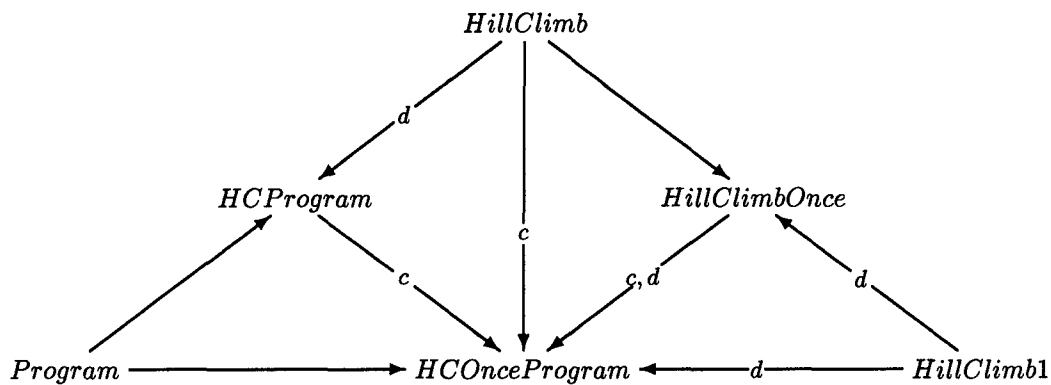
  op MaxTries :  $D \rightarrow \text{Pos}$ 
  definition of MaxTries is
    axiom  $\forall(x : D) (\text{MaxTries}(x) = \text{succ}(\text{zero}))$ 
  end-definition

end-spec

interpretation HillClimbOnce : HillClimb  $\Rightarrow$  HillClimb1 is
  mediator HillClimbOnce
  domain-to-mediator {InitSol  $\rightarrow$  InitSol1}
  codomain-to-mediator import-morphism

```

Figure 87. Refining *HillClimb* for Single-Trial Hill Climbing



spec *HCOncProgram* **is**
colimit of diagram
nodes *HillClimb*, *HCProgram*, *HillClimbOnce*
arcs *HillClimb* $\rightarrow$ *HCProgram* : {}
HillClimb $\rightarrow$ *HillClimbOnce* : {*InitSol* $\rightarrow$ *InitSol1*}
end-diagram
interpretation *HillClimbingOnce* : *Program* $\Rightarrow$ *HillClimb1* **is**
mediator *HCOncProgram*
domain-to-mediator {*O* $\rightarrow$ *LO*, *F* $\rightarrow$ *HillClimbMain*}
codomain-to-mediator {}

Figure 88. Program Scheme for Single-Trial Hill Climbing

```

spec HC1P-Import is
  colimit of diagram
    nodes FLocalSearch, HillClimb1, FLocalOptimum
    arcs FLocalSearch → FLocalOptimum :
      cocone-morphism from FLocalSearch
      FLocalSearch → HillClimb1 : {}
  end-diagram

spec HC1Program is
  import HC1P-Import

  op BetterSol : D, R, R → R
  definition of BetterSol is
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z1) \leq Cost(x, z2) \Rightarrow BetterSol(x, z1, z2) = z1)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow BetterSol(x, z1, z2) = z2)$ 
  end-definition

  op HillClimbMain : D → R
  definition of HillClimbMain is
    axiom  $\forall(x : D) (HillClimbMain(x) = HillClimbStep(x, InitSol(x)))$ 
  end-definition

  op HillClimbStep : D, R → R
  definition of HillClimbStep is
    axiom  $\forall(x : D, z : R) (LocallyOptimal(x, z) \Rightarrow HillClimbStep(x, z) = z)$ 
    axiom  $\forall(x : D, z : R)$ 
       $(\neg LocallyOptimal(x, z)$ 
         $\Rightarrow HillClimbStep(x, z) = HillClimbStep(x, BetterNeighbor(x, z)))$ 
  end-definition

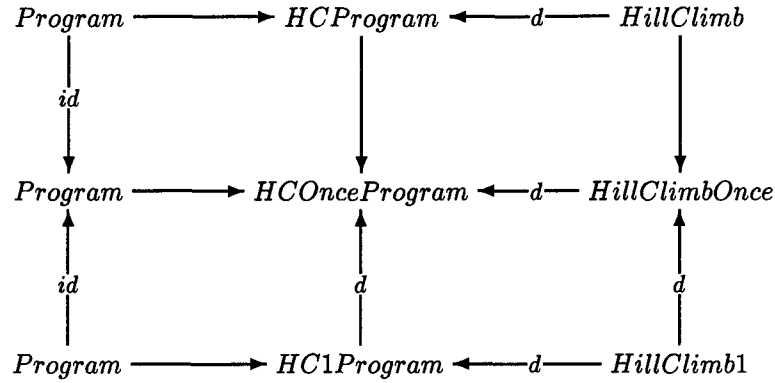
  theorem  $\forall(x : D, z : R) (I(x) \wedge O(x, z) \Rightarrow LO(x, HillClimbStep(x, z)))$ 
  theorem  $\forall(x : D) (I(x) \Rightarrow LO(x, HillClimbMain(x)))$ 
end-spec

interpretation HillClimbing1 : Program  $\Rightarrow$  HillClimb1 is
  mediator HC1Program
  domain-to-mediator  $\{O \rightarrow LO, F \rightarrow HillClimbMain\}$ 
  codomain-to-mediator {}

ip-scheme-morphism FLO-as-HillClimbing1 : FLocalOptimization → HillClimbing1 is
  domain-sm import-morphism
  mediator-sm {}
  codomain-sm {}

```

Figure 89. Simplified Program Scheme for Single-Trial Hill Climbing



ip-scheme-morphism *HC-to-HCOnc* : *HillClimbing* → *HillClimbingOnce* is
domain-sm identity-morphism
mediator-sm cocone-morphism from *HCProgram*
codomain-sm {*InitSol* → *InitSol*}

ip-scheme-morphism *HC1-to-HCOnc* : *HillClimbing1* → *HillClimbingOnce* is
domain-sm identity-morphism
mediator-sm {}
codomain-sm import-morphism

Figure 90. Refining *HillClimbing* to *HillClimbing1*

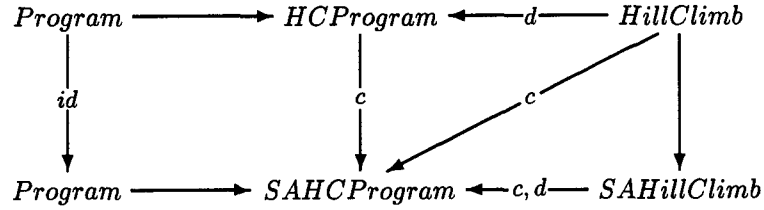
the old *BetterNeighbor*. The axiom for *BestNeighbor* is more specific about which neighbor is returned, but it does not specify how to break ties and thus is not yet fully defined. Interestingly, the precondition that a better neighbor exists is no longer necessary. If a better neighbor exists, selecting a best neighbor will perforce select one that is better than the current solution. Thus, the axiom in *HillClimb* is a theorem in *SAHillClimb*. Since everything else is the same, there is a morphism *HillClimb* → *SAHillClimb*. The interpretation *HillClimbing* is extended to this new target by forming a new mediator spec via pushout and composing arrows to get a morphism to it from *Program*. The result is presented as a new program scheme, *SAHillClimbing*, with an interpretation morphism from *HillClimbing*.

The refinements for single-trial search and steepest ascent can be combined by taking the pushout (in the category of interpretations and interpretation morphisms) of *HC-to-HCOnc* and

HC-to-SAHC. A simplification corresponding to *HillClimbing1* would require some user intervention to get right.

6.3.4 Hill Climbing with Neutral Moves. Selman and his colleagues use another variant on hill climbing in their work on the boolean satisfiability problem (64). In their algorithm they choose the best neighbor at each step, break ties randomly, and most importantly, permit neutral moves, that is, moves to solutions of equal cost. Local optimality is thus no longer the stopping criterion. Instead a combination of criteria are used. First, if the current solution can be shown to be globally optimal, search can stop. Since in general global optimality cannot be determined, a sufficient condition, *GOTest*, is included in the domain theory. This test can effectively be removed if no good test is available by refining it to *false*. Second, if the current solution is strictly better than all of its neighbors, so that in other words there are no improving moves available and no neutral moves available, search can stop. Such a solution is called a *strict local optimum*. Finally, a new input parameter, *MaxNeutral*, is added to the program to set a limit on the number of consecutive neutral moves that can be made. If *MaxNeutral* is set to zero, the result is standard hill climbing. The current solution at the end of the search is the best solution visited, possibly one of several tied for best, and is returned. The algorithm visits several local optima in the course of a single search, with all but possibly the last being non-strict. As with regular hill climbing above, the algorithm performs multiple searches from different initial solutions to visit still more local optima.

Figure 92 shows the domain theory and most of the other components of a program scheme for this algorithm; Figure 93 has the mediator spec. The domain theory imports *HillClimb*, but the program scheme is not a refinement of hill climbing: it is a separate approach to local optimization. It is also not quite Selman's algorithm. It commits to fewer of the design decisions they made and hence is more general. For example, *HillClimbStep* prefers improving moves over neutral ones, but does not choose a best neighbor.



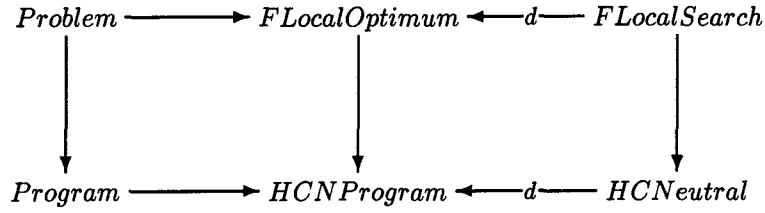
spec SAHillClimb is
import *FLocalSearch*, *Nat*
sort *ISP*
op *LegalISP* : *D*, *ISP* → *Boolean*
op *InitISP* : *D* → *ISP*
axiom $\forall (x : D) (I(x) \Rightarrow \text{LegalISP}(x, \text{InitISP}(x)))$
op *InitSol* : *D*, *ISP* → *R*, *ISP*
axiom $\forall (x : D, p : \text{ISP}) (I(x) \wedge \text{LegalISP}(x, p) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$
axiom $\forall (x : D, p : \text{ISP}) (I(x) \wedge \text{LegalISP}(x, p) \Rightarrow \text{LegalISP}(x, (\text{project } 2)(\text{InitSol}(x, p))))$
op *MaxTries* : *D* → *Pos*
op *BestNeighbor* : *D*, *R* → *R*
axiom $\forall (x : D, z : R) (N(x, z, \text{BetterNeighbor}(x, z)) \wedge \forall (z'' : R) \text{Cost}(x, \text{BetterNeighbor}(x, z)) < \text{Cost}(x, z''))$
end-spec

spec SAHCPProgram is
colimit of diagram
nodes *HillClimb*, *HCProgram*, *SAHillClimb*
arcs *HillClimb* → *HCProgram* : {}
HillClimb → *SAHillClimb* : {*BetterNeighbor* → *BestNeighbor*}
end-diagram

interpretation SAHillClimbing : Program ⇒ SAHillClimb is
mediator *SAHCPProgram*
domain-to-mediator {*O* → *LO*, *F* → *HillClimbMain*}
codomain-to-mediator cocone-morphism from SAHillClimb

ip-scheme-morphism HC-to-SAHC : HillClimbing → SAHillClimbing is
domain-sm identity-morphism
mediator-sm cocone-morphism from HCProgram
codomain-sm {*BetterNeighbor* → *BestNeighbor*}

Figure 91. Hill Climbing Refined to Steepest Ascent Hill Climbing



```

spec HCNeutral is
  import HillClimb

  op MaxNeutral : D → Nat

  op Optimal : D, R → Boolean
  definition of Optimal is
    axiom  $\forall (x : D, z : R) (Optimal(x, z) \Leftrightarrow$ 
       $\forall (z' : R) (O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z')))$ 
    end-definition

  op GOTest : D, R → Boolean
  axiom  $\forall (x : D, z : R) (GOTest(x, z) \Rightarrow Optimal(x, z))$ 
end-spec

spec HCNP-Import is
  colimit of diagram
    nodes FLocalSearch, HCNeutral, FLocalOptimum
    arcs FLocalSearch → FLocalOptimum :
      cocone-morphism from FLocalSearch
      FLocalSearch → HCNeutral : {}
    end-diagram

interpretation NHillClimbing : Program ⇒ HCNeutral is
  mediator HCNProgram
  domain-to-mediator {O → LO, F → HillClimbMain}
  codomain-to-mediator {}

ip-scheme-morphism FLO-to-HCN : FLocalOptimization → NHillClimbing is
  domain-sm import-morphism
  mediator-sm {}
  codomain-sm {}
  
```

Figure 92. Program Scheme for Hill Climbing with Neutral Moves

```

spec HCNProgram is
  import HCNP-Import

  op BetterSol : D, R, R → R
  definition of BetterSol is
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z1) \leq Cost(x, z2) \Rightarrow BetterSol(x, z1, z2) = z1)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow BetterSol(x, z1, z2) = z2)$ 
  end-definition

  op HillClimbMain : D → R
  definition of HillClimbMain is
    axiom  $\forall(x : D) (HillClimbMain(x)$ 
       $= HillClimbIter(x, MaxTries(x), MaxNeutral(x), InitISP(x)))$ 
  end-definition

  op HillClimbIter : D, Pos, Nat, ISP → R
  definition of HillClimbIter is
    axiom  $\forall(x : D, n : Nat, p : ISP, z : R)$ 
       $(z = (\text{project } 1)(InitSol(x, p))$ 
         $\Rightarrow HillClimbIter(x, succ(zero), n, p) = HillClimbStep(x, z, n, zero))$ 
    axiom  $\forall(x : D, k : Pos, n : Nat, p : ISP, z : R, p' : ISP, z' : R, z'' : R)$ 
       $(\langle z, p' \rangle = InitSol(x, p) \wedge z' = HillClimbStep(x, z, n, zero)$ 
         $\wedge z'' = HillClimbIter(x, k, p', n)$ 
         $\Rightarrow HillClimbIter(x, succ(nat-of-pos(k)), p, n) = BetterSol(x, z', z''))$ 
  end-definition

  op HillClimbStep : D, R, Nat, Nat → R
  definition of HillClimbStep is
    axiom  $\forall(x : D, z : R, max : Nat, step : Nat)$ 
       $(\forall(z' : R) (N(x, z, z') \Rightarrow Cost(x, z) < Cost(x, z')) \vee GOTest(x, z)$ 
         $\vee max < step \vee (LocallyOptimal(x, z) \wedge max = step)$ 
         $\Rightarrow HillClimbStep(x, z, max, step) = z)$ 
    axiom  $\forall(x : D, z : R, max : Nat, step : Nat)$ 
       $(\neg LocallyOptimal(x, z) \wedge step \leq max$ 
         $\Rightarrow HillClimbStep(x, z, max, step)$ 
         $= HillClimbStep(x, BetterNeighbor(x, z), max, zero))$ 
    axiom  $\forall(x : D, z : R, max : Nat, step : Nat)$ 
       $(LocallyOptimal(x, z) \wedge \neg GOTest(x, z) \wedge step < max$ 
         $\wedge \exists(z' : R) (N(x, z, z') \wedge Cost(x, z) = Cost(x, z'))$ 
         $\Rightarrow \exists(z' : R) (N(x, z, z') \wedge Cost(x, z) = Cost(x, z')$ 
           $\wedge HillClimbStep(x, z, max, step)$ 
           $= HillClimbStep(x, z', max, nat-of-pos(succ(step))))$ 
  end-definition

  theorem  $\forall(x : D, z : R, max : Nat) (I(x) \wedge O(x, z) \Rightarrow LO(x, HillClimbStep(x, z, max, zero)))$ 
end-spec

```

Figure 93. Mediator Spec for Hill Climbing with Neutral Moves

A more significant difference is in the interpretation of *MaxNeutral*. In Selman's algorithm, this parameter limits the total number of moves made. If this parameter is too small, the algorithm will not return a local optimum because it will not have time to reach the first one. By using this parameter as the maximum number of steps since the last improving move, the algorithm is guaranteed to run long enough to reach a local optimum. Once the first local optimum has been reached, the search makes neutral moves (either randomly as Selman does or by some other means) until the limit is met or an improving move is found. If the latter, it then climbs to a new local optimum.

Theorem 6.3.4 *In the spec HNCProgram,*

$$\forall(x : D, z : R, max : Nat) (I(x) \wedge O(x, z) \Rightarrow LO(x, HillClimbStep(x, z, max, zero)))$$

Proof. Let x and z satisfy $I(x) \wedge O(x, z)$. If z is not locally optimal, then an improving move exists. Since the step parameter is zero, $step \leq max$. By the second axiom, a better neighbor is chosen and search continues from there. Since the neighborhood is feasible, this new solution is feasible. This argument can be applied recursively to show that the operation always climbs to a local optimum. As long as z is not locally optimal, only the second axiom applies: z cannot be globally optimal and the step parameter is maintained at zero.

If z is locally optimal, then either the first or third axiom might apply. If any of the conditions of the first axiom are satisfied, z is returned. The first axiom checks for strict local optimality, which is equivalent to there being no neutral moves available. If neutral moves are available and the current solution is not globally optimal and the step parameter permits, then the third axiom applies. It chooses a neighbor of the same cost and increases the step parameter by one. Eventually either $step$ will exceed max and the search will terminate via the first axiom or an improving move will be found. If the latter, then that move will be selected and the step parameter will be set back

to zero. By the previous argument, search will continue to a new local optimum before increasing the step parameter.

Note that if $max = zero$ then the third axiom never applies and as soon as the first local optimum is found, the first axiom will return it. $\square$

Proofs concerning the behavior of *HillClimbIter* and *HillClimbMain* are essentially the same as they were for *HillClimbing*.

6.3.5 Simulated Annealing. The elements of simulated annealing were explained in Chapter I and a generic algorithm was presented. The essential characteristics of simulated annealing are in the move selection rules. At each step a neighbor is chosen arbitrarily (usually randomly) and its cost compared to the current solution. If the neighbor is better or the same, the move is accepted. If the neighbor is worse, the move may still be accepted. This decision is influenced by the magnitude of the change in cost and by a search parameter called *temperature*. At high temperatures worsening moves are likely to be accepted, while at low temperatures they are not. Thus at high temperatures the search resembles a random walk through the solution space, while at low temperatures it resembles hill climbing. At intermediate temperatures the search is attracted to local optima, but it still makes some worsening moves. Search begins with the temperature set relatively high and gradually reduces it. Search ends when it is determined heuristically to be *frozen*, meaning further improvement is highly unlikely. Some algorithms return the final solution, while others maintain a record of the best solution seen during the search and return that.

Simulated annealing requires a feasible neighborhood to guarantee that a feasible solution is returned. Reachability is highly recommended (14). Under fairly general assumptions, its asymptotic or expected behavior is to return a global optimum (14), but the reasoning behind such results is beyond the capabilities of most theorem provers and in practice the settings required to achieve guaranteed results yield run times that are much too high. Since the neighborhood of a solution is typically not searched systematically, improving moves can be missed (though ideally with very

low probability); thus the solution returned may not even be locally optimal. The only claim we shall make is that the algorithm returns a feasible solution.

Two program schemes for simulated annealing will be provided. The first is very abstract and attempts to describe the broadest class of algorithms that can reasonably be considered to be simulated annealing. The second is pretty close to the one in Chapter I.

Figures 94 and 95 present a domain theory for abstract simulated annealing. The theory introduces the sort that will be used for temperatures, *Temp*, as a total order. The state of a search is described by the spec *State*, with operations to set and retrieve the current temperature. This spec does not preclude the possibility that states may contain other information as well.

The codomain of the cost function for generic optimization problems, *R*, is required only to be a total order. Simulated annealing requires in addition a difference operation, in order to compare solutions more precisely. Ordinary subtraction is most often used, but this is not the only possibility. The magnitude of improving moves is not considered, so the kind of difference operation sometimes defined for natural numbers, where all normal subtractions that yield negative numbers are instead defined to be zero, is sufficient. Even division can be suitable if relative changes are judged more significant than absolute ones for a particular problem. The spec *Difference* characterizes some basic properties of difference operations over total orders. It is broad enough to encompass all the cases just considered and specific enough to do what we need.

Next a colimit is used to incorporate state, temperature, a difference operation and natural numbers into local search over a feasible neighborhood. Spec *SimAnneal* imports this colimit and defines the remaining sorts and operations needed. Computing multiple initial starting solutions is done by the sort *ISP* and operations *InitISP* and *InitSol*, as for hill climbing. The predicate *LegalISP* has been dropped to illustrate a different style. If a refinement of *SimAnneal* uses a sort for *ISP* that includes illegal values, then a subsort will have to be used. *MaxTries* is a tunable parameter of the overall strategy, representing how many searches to carry out. The sort *SP*


```

spec Temp is
  translate TotalOrder by  $\{E \rightarrow Temp\}$ 

spec SASState is
  import Temp

  sort State

  op Temp : State  $\rightarrow$  Temp

  op SetTemp : State, Temp  $\rightarrow$  State
  axiom  $\forall (s : State, t : Temp) (Temp(SetTemp(s, t)) = t)$ 
end-spec

spec Difference is
  import TotalOrder

  op  $_ - _ : E, E \rightarrow E$ 

  axiom  $\forall (a : E, b : E) ((a - b) \leq (b - a) \Rightarrow a \leq b)$ 
  axiom  $\forall (a : E, b : E, c : E, d : E) (a \leq b \wedge c \leq d \Rightarrow (c - b) \leq (d - a))$ 
  axiom  $\forall (a : E, b : E, c : E) ((a - b) \leq c \Rightarrow (a - c) \leq b)$ 
  axiom  $\forall (a : E, b : E) (a - (b - b) = a)$ 

  theorem  $\forall (a : E, b : E) (a - a = b - b)$ 
  theorem  $\forall (a : E, b : E, c : E) (a - b = c \Rightarrow (a - c) \leq b)$ 
  theorem  $\forall (a : E, b : E, c : E) (b \leq c \Rightarrow (b - a) \leq (c - a))$ 
  theorem  $\forall (a : E, b : E, c : E) (b \leq c \Rightarrow (a - c) \leq (a - b))$ 
  theorem  $\forall (a : E, b : E) (a \leq b \Rightarrow (a - b) \leq (b - a))$ 
  theorem  $\forall (a : E, b : E) (a - b = b - a \Rightarrow a = b)$ 
end-spec

spec SA-Import is
  translate colimit of diagram
    nodes TotalOrder, FLocalSearch, Difference, SASState, Nat
    arcs TotalOrder  $\rightarrow$  FLocalSearch :  $\{E \rightarrow \mathcal{R}\}$ 
           TotalOrder  $\rightarrow$  Difference :  $\{\}$ 
  end-diagram by  $\{E \rightarrow \mathcal{R}\}$ 

```

Figure 94. Domain Theory for Abstract Simulated Annealing

```

spec SimAnneal is
  import SA-Import

  sort ISP
  op InitISP : D → ISP

  op InitSol : D, ISP → R, ISP
  axiom  $\forall(x : D, p : ISP) (I(x) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$ 

  op MaxTries : D → Pos

  sort SP
  op InitSP : D → SP

  op InitState : D, SP → State, SP
  op NewState : D, R, R, State, SP → State, SP
  axiom  $\forall(x : D, z : R, z' : R, s : State, sp : SP)$ 
     $(Temp((\text{project } 1)(\text{NewState}(x, z, z', s, sp))) \leq Temp(s))$ 
  op frozen : D, R, State → Boolean

  op GetNeighbor : D, R, SP → R, SP
  axiom  $\forall(x : D, z : R, sp : SP, z' : R, sp' : SP)$ 
     $(I(x) \wedge O(x, z) \wedge \langle z', sp' \rangle = \text{GetNeighbor}(x, z, sp) \Rightarrow N(x, z, z'))$ 

  op AcceptMove : D, R, R, Temp, SP → Boolean, SP
  axiom  $\forall(x : D, y_1 : R, y_2 : R, t : Temp, sp : SP)$ 
     $(y_2 \leq y_1 \Rightarrow (\text{project } 1)(\text{AcceptMove}(x, y_1, y_2, t, sp)))$ 
  axiom  $\forall(x : D, y_1 : R, y_2 : R, t_1 : Temp, t_2 : Temp, sp : SP)$ 
     $(t_1 \leq t_2$ 
       $\Rightarrow ((\text{project } 1)(\text{AcceptMove}(x, y_1, y_2, t_2, sp)))$ 
       $\Rightarrow (\text{project } 1)(\text{AcceptMove}(x, y_1, y_2, t_1)))$ )
  axiom  $\forall(x : D, y_1 : R, y_2 : R, y_3 : R, y_4 : R, t : Temp, sp : SP)$ 
     $((y_2 - y_1) = (y_4 - y_3)$ 
       $\Rightarrow ((\text{project } 1)(\text{AcceptMove}(x, y_1, y_2, t, sp)))$ 
       $\Leftrightarrow (\text{project } 1)(\text{AcceptMove}(x, y_3, y_4, t, sp))))$ 
  axiom  $\forall(x : D, y_1 : R, y_2 : R, y_3 : R, y_4 : R, t : Temp, sp : SP)$ 
     $((y_2 - y_1) \leq (y_4 - y_3)$ 
       $\Rightarrow ((\text{project } 1)(\text{AcceptMove}(x, y_1, y_2, t, sp)))$ 
       $\Rightarrow (\text{project } 1)(\text{AcceptMove}(x, y_3, y_4, t, sp))))$ 

  op Optimal : D, R → Boolean
  definition of Optimal is
    axiom  $\forall(x : D, z : R) (\text{Optimal}(x, z) \Leftrightarrow$ 
       $\forall(z' : R) (O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z')))$ 
  end-definition

  op GOTest : D, R → Boolean
  axiom  $\forall(x : D, z : R) (\text{GOTest}(x, z) \Rightarrow \text{Optimal}(x, z))$ 
end-spec

```

Figure 95. Domain Theory for Abstract Simulated Annealing, Cont.

represents *selectionparameters* that are needed each time the algorithm needs to make a selection. The most common interpretation of this sort is that it is the seed of a random number generator. All operations that potentially include an arbitrary or random choice will accept a parameter of sort *SP* and return one as well. If a future design choice eliminates the need for this parameter in some operations, it is easily removed.

Operations on states include *InitState*, *NewState* and *frozen*. Since all we know about states is that they include temperatures, we can only address the effects of operations on states in terms of their effects on temperatures. Indeed, there is little enough to say. *InitState* sets the initial temperature, but a universal method for doing so effectively is not known, so nothing is specified. *NewState* may change the temperature, under the restriction that it may only be reduced, never increased. Some varieties of simulated annealing do in fact sometimes increase the temperature, so these have been pared away: a separate scheme would be required to describe them. The predicate *frozen* is again unspecified except for its signature: the methods used in practice are just too broad to describe abstractly.

The operations *GetNeighbor* and *AcceptMove* generate moves and select among them, respectively. Both involve potentially random choices, so both include *SP* in their signatures. The axiom for *GetNeighbor* states that it does return a neighbor of the current solution, and nothing more. This is consistent with typical implementations of simulated annealing: the choice of neighbor is completely random. The properties of *AcceptMove* are described a bit more specifically. The signature says that acceptance depends only on the costs of the solutions, not any other properties, and on the temperature, not any other aspects of search state. The first axiom states that all improving or neutral moves are accepted, regardless of temperature. The second axiom says that at higher temperatures moves are more likely to be accepted. Since even random choices are only pseudo-random, the behavior of *AcceptMove* on a given set of inputs is deterministic. Thus likelihood properties can be expressed by fixing a selection parameter, *sp*, and comparing the

results when other parameters are allowed to vary. The third axiom says that acceptance is based on the *difference* in cost, not their magnitudes. Thus if two distinct moves involve the same cost difference, their likelihood of acceptance at a given temperature is the same. Finally, the fourth axiom says that worsening moves of smaller magnitude are more likely to be accepted than larger ones.

Optimal and *GOTest* are used as they were for hill climbing with neutral moves, to provide an alternative stopping criterion.

Figure 96 shows the program spec or mediator for simulated annealing. The general form should be familiar. The main routine calls an iterator to make a specified number of searches and return the best solution found. The iterator initializes each search and calls a step function. The step function quits if the current search state is frozen or if the current solution passes the test for global optimality. Otherwise it generates a move, decides whether to accept it or not, updates the state accordingly and recurses. The correctness of the main routine follows directly from the correctness of the iterator, and that of the iterator from that of the step function. Unfortunately, we do not know enough to prove the correctness of the step function. In particular, there is no way to prove that the recursion terminates, that is, that a frozen state ever occurs. This follows from the fact that *frozen* has no axioms in the domain theory. Even calling the axioms of *SimAnnealStep* a definition assumes termination and so is technically not allowed. Further work on characterizing termination of simulated annealing is needed.

Figure 97 presents the complete program scheme for abstract simulated annealing. The problem solved is not local optimization, but merely the problem of finding a feasible solution in the presence of a neighborhood structure. *FeasibleSolution* is a specification for this class of problems. It is simply the cocone morphism from *Problem* to *FLocalSearch*, raised to an interpretation. *SimAnnealing* is the program specification; again, the morphism $Program \rightarrow S A Program$ is not verifiable because the correctness axiom cannot be proved. One could substitute *Problem*

```

spec SAProgram is
  import SimAnneal

  op BetterSol : D, R, R → R
  definition of BetterSol is
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z1) \leq Cost(x, z2) \Rightarrow BetterSol(x, z1, z2) = z1)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow BetterSol(x, z1, z2) = z2)$ 
  end-definition

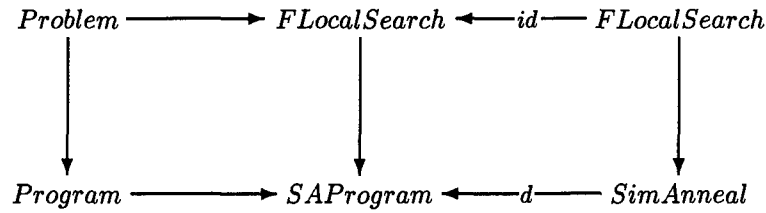
  op SimAnnealMain : D → R
  definition of SimAnnealMain is
    axiom  $\forall(x : D) (SimAnnealMain(x)$ 
       $= (\text{project } 1)(SimAnnealIter(x, MaxTries(x), InitISP(x), InitSP(x))))$ 
    end-definition

  op SimAnnealIter : D, Pos, ISP, SP → R, SP
  definition of SimAnnealIter is
    axiom  $\forall(x : D, p : ISP, sp : SP, z : R, s : State, sp' : SP)$ 
       $(z = (\text{project } 1)(InitSol(x, p)) \wedge \langle s, sp' \rangle = InitState(x, sp))$ 
       $\Rightarrow SimAnnealIter(x, succ(zero), p, sp)$ 
       $= SimAnnealStep(x, z, s, sp')$ 
    axiom  $\forall(x : D, k : Pos, p : ISP, sp : SP, z : R, p' : ISP, s : State, sp' : SP, z' : R,$ 
       $z'' : R, p'' : ISP, sp'' : SP)$ 
       $(\langle z, p' \rangle = InitSol(x, p) \wedge \langle s, sp' \rangle = InitState(x, sp))$ 
       $\wedge \langle z', sp'' \rangle = SimAnnealStep(x, z, s, sp')$ 
       $\wedge \langle z'', sp''' \rangle = SimAnnealIter(x, k, p', sp'')$ 
       $\Rightarrow SimAnnealIter(x, succ(nat-of-pos(k)), p, sp)$ 
       $= \langle BetterSol(x, z', z''), sp''' \rangle$ 
    end-definition

  op SimAnnealStep : D, R, State, SP → R, SP
  definition of SimAnnealStep is
    axiom  $\forall(x : D, z : R, s : State, sp : SP)$ 
       $(frozen(x, z, s) \vee GOSTest(x, z) \Rightarrow SimAnnealStep(x, z, s, sp) = \langle z, sp \rangle)$ 
    axiom  $\forall(x : D, z : R, s : State, sp : SP, z' : R, sp' : SP, b : Boolean, sp'' : SP,$ 
       $s' : State, sp''' : SP)$ 
       $(\neg frozen(x, z, s) \wedge \neg GOSTest(x, z) \wedge \langle z', sp' \rangle = GetNeighbor(x, z, sp)$ 
       $\wedge \langle b, sp'' \rangle = AcceptMove(x, Cost(x, z), Cost(x, z'), Temp(s), sp')$ 
       $\wedge b \wedge \langle s', sp''' \rangle = NewState(x, z, z', s, sp''))$ 
       $\Rightarrow SimAnnealStep(x, z, s, sp) = SimAnnealStep(x, z', s', sp''')$ 
    axiom  $\forall(x : D, z : R, s : State, sp : SP, z' : R, sp' : SP, b : Boolean, sp'' : SP,$ 
       $s' : State, sp''' : SP)$ 
       $(\neg frozen(x, z, s) \wedge \neg GOSTest(x, z) \wedge \langle z', sp' \rangle = GetNeighbor(x, z, sp)$ 
       $\wedge \langle b, sp'' \rangle = AcceptMove(x, Cost(x, z), Cost(x, z'), Temp(s), sp')$ 
       $\wedge \neg b \wedge \langle s', sp''' \rangle = NewState(x, z, z, s, sp''))$ 
       $\Rightarrow SimAnnealStep(x, z, s, sp) = SimAnnealStep(x, z, s', sp''')$ 
    end-definition
end-spec

```

Figure 96. Mediator Spec for Abstract Simulated Annealing



interpretation *FeasibleSolution* : *Problem* $\Rightarrow$ *FLocalSearch* is
mediator *FLocalSearch*
domain-to-mediator cocone-morphism from *Problem*
codomain-to-mediator identity-morphism

interpretation *SimAnnealing* : *Program* $\Rightarrow$ *SimAnneal* is
mediator *SAProgram*
domain-to-mediator $\{F \rightarrow \text{SimAnnealMain}\}$
codomain-to-mediator import-morphism

ip-scheme-morphism *Feas-to-SA* : *FeasibleSolution* $\rightarrow$ *SimAnnealing* is
domain-sm $\{\}$
mediator-sm $\{\}$
codomain-sm $\{\}$

Figure 97. Program Scheme for Abstract Simulated Annealing

for *Program* and treat abstract simulated annealing as a refined problem class rather than as a program scheme, using *SimAnnealMain* to refine *O*, but it is left as a program scheme in the hope that the termination problem will be solved in the future.

The second program scheme for simulated annealing that will be presented refines the abstract specification in a number of ways. Like the generic algorithm presented in Chapter I, this one is based on executing a series of *runs*. During a run, the temperature is held constant. At the end of each run, the temperature is reduced by some constant factor called the *cooling ratio*. A run can be terminated prematurely if the ratio of accepted moves to moves, called the *acceptance ratio*, is too high: such a situation suggests that the temperature is too high and so the search is too random. A run in which the acceptance ratio is too low suggests the search may be frozen. Search is terminated after some number of consecutive runs of this sort.

Figures 98 through 100 present the domain theory for run-based simulated annealing. It assumes the existence of a spec for real numbers and mathematical operations on them. It also assumes the existence of a random number generator capable of producing uniformly distributed pseudo-random numbers over an identified interval. Developing detailed specs for these concepts is beyond the scope of this dissertation.

The spec *RLSAState* imports the state spec from the generic theory and extends it with several new attributes. *RunLength* is the number of steps or moves made so far in the current run, including moves that were considered but not accepted. *NAccepted* is the number of moves that were accepted. *NFrozen* is the number of consecutive runs that have appeared frozen. The operations with these names extract the corresponding values from a state variable. Each attribute also has an operation that returns a state with a particular attribute value. Axioms are included that insure that set values are returned and that each set operation does not affect any of the values set by the others. This spec still does not preclude the possibility of later refinements adding still more attributes. The approach becomes unwieldy, however, with more than about five attributes: the n -th attribute requires n axioms to describe.

The method to be used for accepting moves assumes that the sorts *Temp* and $\mathcal{R}$ are both *Real*, so *RLSA-Import* combines local search, states, real numbers and random number generation to carry this out. The spec *RLSimAnneal* introduces the usual sort and operations for finding initial solutions. The search parameter sort is *Seed*, since we are committing to random selection.

Run-based simulated annealing makes use of a large number of tunable parameters, represented by operations on input values. *MaxTries* is again the number of searches to run. *MinRunLength* is the minimum number of steps in a run. The acceptance ratio will not be considered until at least this many moves have been made in the current run. *MaxRunLength* is the maximum number of steps in a run. *MinAcceptRate* is the acceptance ratio threshold below which a run is considered frozen. *MaxAcceptRate* is the acceptance ratio threshold above which

```

spec Real is sorts Real, RPos ... end-spec

spec RealMath is import Real ... end-spec

spec Random is
  import RealMath
  sort Seed
  op Uniform : Seed, Real, Real → Seed, Real
  axiom  $\forall (s : \text{Seed}, \min : \text{Real}, \max : \text{Real})$ 
     $(\min \leq (\text{project } 2)(\text{Uniform}(s, \min, \max)))$ 
     $\wedge (\text{project } 2)(\text{Uniform}(s, \min, \max)) \leq \max$ 
  ...
end-spec

spec RLState is
  import SState, Nat
  op RunLength : State → Nat
  op SetRunLength : State, Nat → State
  axiom  $\forall (s : \text{State}, rl : \text{Nat}) (\text{RunLength}(\text{SetRunLength}(s, rl)) = rl)$ 
  axiom  $\forall (s : \text{State}, rl : \text{Nat}, t : \text{Temp})$ 
     $(\text{SetRunLength}(\text{SetTemp}(s, t), rl) = \text{SetTemp}(\text{SetRunLength}(s, rl), t))$ 

  op NAccepted : State → Nat
  op SetNAccepted : State, Nat → State
  axiom  $\forall (s : \text{State}, na : \text{Nat}) (\text{NAccepted}(\text{SetNAccepted}(s, na)) = na)$ 
  axiom  $\forall (s : \text{State}, na : \text{Nat}, t : \text{Temp})$ 
     $(\text{SetNAccepted}(\text{SetTemp}(s, t), na) = \text{SetTemp}(\text{SetNAccepted}(s, na), t))$ 
  axiom  $\forall (s : \text{State}, na : \text{Nat}, rl : \text{Nat})$ 
     $(\text{SetNAccepted}(\text{SetRunLength}(s, rl), na) = \text{SetRunLength}(\text{SetNAccepted}(s, na), rl))$ 

  op NFrozen : State → Nat
  op SetNFrozen : State, Nat → State
  axiom  $\forall (s : \text{State}, nf : \text{Nat}) (\text{NFrozen}(\text{SetNFrozen}(s, nf)) = nf)$ 
  axiom  $\forall (s : \text{State}, nf : \text{Nat}, t : \text{Temp})$ 
     $(\text{SetSetNFrozen}(\text{SetTemp}(s, t), nf) = \text{SetTemp}(\text{SetNFrozen}(s, nf), t))$ 
  axiom  $\forall (s : \text{State}, nf : \text{Nat}, rl : \text{Nat})$ 
     $(\text{SetNFrozen}(\text{SetRunLength}(s, rl), nf) = \text{SetRunLength}(\text{SetNFrozen}(s, nf), rl))$ 
  axiom  $\forall (s : \text{State}, nf : \text{Nat}, na : \text{Nat})$ 
     $(\text{SetNFrozen}(\text{SetNAccepted}(s, na), nf) = \text{SetNAccepted}(\text{SetNFrozen}(s, nf), na))$ 
end-spec

spec RLState-Import is
  translate colimit of diagram
    nodes Triv, Random, RLState, FLocalSearch
    arcs Triv → Random :  $\{E \rightarrow \text{Real}\}$ , Triv → RLState :  $\{E \rightarrow \text{Temp}\}$ ,
        Triv → FLocalSearch :  $\{E \rightarrow \mathcal{R}\}$ 
  end-diagram by  $\{E \rightarrow \text{Real}\}$ 

```

Figure 98. Domain Theory for Run-Length-Based Simulated Annealing


```

spec RLSimAnneal is
  import RLSA-Import

  sort ISP
  op InitISP : D → ISP

  op InitSol : D, ISP → R, ISP
  axiom  $\forall (x : D, p : ISP) (I(x) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$ 

  op MaxTries : D → Pos
  op MinRunLength : D → Pos
  op MaxRunLength : D → Pos
  op MinAcceptRate : D → Real
  op MaxAcceptRate : D → Real
  op MaxFrozen : D → Pos
  op CoolingRatio : D → Real

  axiom  $\forall (x : D) (MinRunLength(x) \leq MaxRunLength(x))$ 
  axiom  $\forall (x : D) (zero < MinAcceptRate(x))$ 
  axiom  $\forall (x : D) (MinAcceptRate(x) \leq MaxAcceptRate(x))$ 
  axiom  $\forall (x : D) (MaxAcceptRate(x) \leq one)$ 
  axiom  $\forall (x : D) (zero \leq CoolingRatio(x))$ 
  axiom  $\forall (x : D) (CoolingRatio(x) < one)$ 

  op InitSeed : D → Seed

  op InitState : D, Seed → State, Seed
  axiom  $\forall (x : D) (RunLength((\text{project } 1)(\text{InitState}(x))) = zero)$ 
  axiom  $\forall (x : D) (NAccepted((\text{project } 1)(\text{InitState}(x))) = zero)$ 
  axiom  $\forall (x : D) (NFrozen((\text{project } 1)(\text{InitState}(x))) = zero)$ 
  axiom  $\forall (x : D) (zero \leq Temp((\text{project } 1)(\text{InitState}(x))))$ 

  op RecordMove : D, R, R, State → State
  definition of RecordMove is
    axiom  $\forall (x : D, z : R, z' : R, s : State)$ 
       $(z = z' \Rightarrow RecordMove(x, z, z', s)$ 
         $= SetRunLength(s, nat-of-pos(succ(RunLength(s))))$ 
      )
    axiom  $\forall (x : D, z : R, z' : R, s : State)$ 
       $(\neg(z = z') \Rightarrow RecordMove(x, z, z', s)$ 
         $= SetNAccepted(SetRunLength(s, nat-of-pos(succ(RunLength(s))))$ 
           $nat-of-pos(succ(NAccepted(s))))$ 
      )
  end-definition

  op EndOfRun : D, State → Boolean
  definition of EndOfRun is
    axiom  $\forall (x : D, s : State) (EndOfRun(x, s) \Leftrightarrow$ 
       $nat-of-pos(MaxRunLength(x)) \leq RunLength(s)$ 
       $\vee (nat-of-pos(MinRunLength(x)) \leq RunLength(s)$ 
         $\wedge \forall (rl : Pos) (RunLength(s) = nat-of-pos(rl)$ 
           $\Rightarrow MaxAcceptRate(x) < real-of-nat(NAccepted(s))/rpos-of-pos(rl))))$ 
    )
  end-definition

```

Figure 99. Domain Theory for Run-Length-Based Simulated Annealing, Cont.

```

op FinishRun : D, State → State
definition of FinishRun is
  axiom  $\forall (x : D, s : State)$ 
    (nonzero?(RunLength(s))
       $\wedge \forall (rl : Pos) (RunLength(s) = nat-of-pos(rl)$ 
         $\Rightarrow real-of-nat(NAccepted(s)) / rpos-of-pos(rl) < MinAcceptRate(x)$ 
         $\Rightarrow FinishRun(x, s) = SetNFrozen(SetNAccepted(SetRunLength(SetTemp(s,$ 
          Temp(s) · CoolingRatio), zero), zero), nat-of-pos(succ(NFrozen(s))))))
  axiom  $\forall (x : D, s : State)$ 
    (RunLength(s) = zero
       $\vee \forall (rl : Pos) (RunLength(s) = nat-of-pos(rl)$ 
         $\Rightarrow MinAcceptRate(x) \leq real-of-nat(NAccepted(s)) / rpos-of-pos(rl)$ 
         $\Rightarrow FinishRun(x, s) = SetNFrozen(SetNAccepted(SetRunLength(SetTemp(s,$ 
          Temp(s) · CoolingRatio), zero), zero), zero))
end-definition

op NewState : D, R, R, State → State
axiom  $\forall (x : D, z : R, z' : R, s : State, s' : State)$ 
  (s' = RecordMove(x, z, z', s)  $\wedge$  EndOfRun(x, s')
     $\Rightarrow NewState(x, z, z', s) = FinishRun(x, s')$ )
axiom  $\forall (x : D, z : R, z' : R, s : State, s' : State)$ 
  (s' = RecordMove(x, z, z', s)  $\wedge \neg EndOfRun(x, s') \Rightarrow NewState(x, z, z', s) = s'$ )

op frozen : D, R, State → Boolean
axiom  $\forall (x : D, z : R, s : State) (nat-of-pos(MaxFrozen(x)) \leq NFrozen(s) \Rightarrow frozen(x, s))$ 

op GetNeighbor : D, R, Seed → R, Seed
axiom  $\forall (x : D, z : R, sp : Seed, z' : R, sp' : Seed)$ 
  (I(x)  $\wedge$  O(x, z)  $\wedge \langle z', sp' \rangle = GetNeighbor(x, z, sp) \Rightarrow N(x, z, z')$ )

op Optimal : D, R → Boolean
definition of Optimal is
  axiom  $\forall (x : D, z : R) (Optimal(x, z) \Leftrightarrow \forall (z' : R) (O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z')))$ 
end-definition

op GOTest : D, R → Boolean
axiom  $\forall (x : D, z : R) (GOTest(x, z) \Rightarrow Optimal(x, z))$ 
end-spec

```

Figure 100. Domain Theory for Run-Length-Based Simulated Annealing, Cont.

the temperature of a run is considered too high. *MaxFrozen* is the threshold for *NFrozen* above which the search is considered frozen. Finally, *CoolingRatio* is the proportion by which the temperature is reduced at the end of each run. The axioms for these parameters define the minimum requirements for correctness of the algorithm to be developed; they do not attempt to describe default, typical or suggested values.

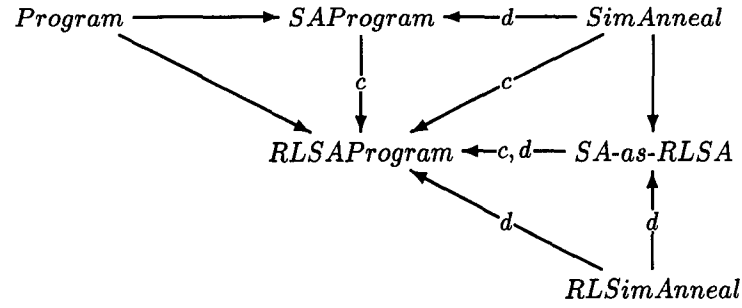
The *InitState* operation is still not fully defined, but axioms assert that the initial run length, number of accepted moves and number of frozen runs are all zero, and the initial temperature is non-negative. Updating the state is described by four operations. *RecordMove* records the effect of a move on the state by incrementing the run length, and also the number of accepted moves if the new solution is distinct from the old one. The predicate *EndOfRun* determines whether a run should end, either because *MaxRunLength* has been met or the acceptance ratio is above *MaxAcceptRate*. *FinishRun* updates the state at the end of a run to begin the next one, if any. It checks whether the run was frozen, resets the run length and number of accepted moves, and reduces the temperature. *NewState* coordinates these three actions by recording each move, checking whether the result satisfies the criteria for ending the run, and if so finishing the run. Note the care with which *EndOfRun* and *FinishRun* compute the acceptance ratio. The division operation for real numbers (or any other field) is not defined when the divisor is zero, so a subsort *RPos* is used in the signature. Applying division then requires the standard technique using relaxation for getting a suitable value of this sort.

The predicate *frozen* returns *true* if the number of frozen runs meets or exceeds *MaxFrozen*. There may be other criteria for freezing as well, so this is not a definition.

GetNeighbor and *GOTest* are no more defined than for abstract simulated annealing. A definition for *AcceptMove* is available, so this operation has been moved out of the domain theory. The definitions that were not moved are all needed by the axioms of some other operation that is not fully defined.

The operations defined in *RLSimAnneal* are all perfectly suited for the definitions in *SAProgram*, by design. Figure 101 shows how *RLSimAnneal* is a refinement of *SimAnneal*, and therefore how the two refinements can be composed to reuse the program spec *SAProgram*. The mediator spec, *SA-as-RLSA*, contains the definition for *AcceptMove*. To be a definition, it must work for negative temperatures even though it will never be used this way. At negative and zero temperatures, then, only improving and neutral moves are accepted. For positive temperatures, acceptance is exponentially distributed based on the cost difference divided by the temperature. For improving and neutral moves, the exponential term is at least one, so all such moves are accepted. The mediator also defines an operation *NewState1* that has the signature required by *NewState* in *SimAnneal* and calls *NewState* in *RLSimAnneal*, returning the same seed value that it was passed. It is much easier to remove unneeded elements of a signature in this way than to add them. The domain morphism of the refinement adjusts the names to match, including *Temp* and $\mathcal{R}$ to *Real*. Normal subtraction over the real numbers satisfies all the properties required by *Difference*.

The composed interpretation *RLSimAnnealing* suffers from the same flaw as *SimAnnealing*: the *frozen* predicate is still not defined well enough to prove termination. In particular, since neutral moves are always accepted, it is possible to spend an infinite number of runs moving among solutions of equal cost without ever escaping from the region. This behavior can be tuned away if it can be shown that the proportion of equal-cost neighbors to neighbors is always less than *MinAcceptRate*, but this is not always an easy proportion to determine. Techniques that cap the total number of steps or runs would solve the problem, but such parameters are difficult to set in practice. Techniques such as capping the number of steps or moves since the last improvement have the potential for interfering with early runs, when the temperature is high and worsening moves common. It seems best to defer these decisions.



```

spec SA-as-RLSA is
  import RLSimAnneal

  op NewState1 : D, R, R, State, Seed → State, Seed
  definition of NewState1 is
    axiom  $\forall(x : D, z : R, z' : R, s : State, sd : Seed)$ 
       $(NewState1(x, z, z', s, sd) = \langle NewState(x, z, z', s), sd \rangle)$ 
  end-definition

  op AcceptMove : D, Real, Real, Real, Seed → Boolean, Seed
  definition of AcceptMove is
    axiom  $\forall(x : D, y_1 : Real, y_2 : Real, t : Real, sd : Seed)$ 
       $(t \leq zero \Rightarrow AcceptMove(x, y_1, y_2, t, sd) = \langle y_2 \leq y_1, sd \rangle)$ 
    axiom  $\forall(x : D, y_1 : Real, y_2 : Real, t : Real, sd : Seed, tpos : RPos, sd' : Seed, u : Real)$ 
       $(zero < t \wedge t = real-of-rpos(tpos) \wedge \langle sd', u \rangle = Uniform(sd, zero, one)$ 
         $\Rightarrow AcceptMove(x, y_1, y_2, t, sd) = \langle u \leq exp((y_1 - y_2)/tpos), sd' \rangle)$ 
  end-definition

end-spec

interpretation SA-to-RLSA : SimAnneal  $\Rightarrow$  RLSimAnneal is
  mediator SA-as-RLSA
  domain-to-mediator
     $\{\mathcal{R} \rightarrow Real, Temp \rightarrow Real, SP \rightarrow Seed, InitSP \rightarrow InitSeed, NewState \rightarrow NewState1\}$ 
  codomain-to-mediator import-morphism

spec RLSAProgram is
  translate colimit of diagram
    nodes SimAnneal, SA-as-RLSA, SAProgram
    arcs SimAnneal  $\rightarrow$  SA-as-RLSA :  $\{\mathcal{R} \rightarrow Real, Temp \rightarrow Real, SP \rightarrow Seed,$ 
       $InitSP \rightarrow InitSeed, NewState \rightarrow NewState1\}$ 
      SimAnneal  $\rightarrow$  SAProgram : import-morphism
  end-diagram by  $\{Real \rightarrow Real, NewState1 \rightarrow NewState1\}$ 

interpretation RLSimAnnealing : Program  $\Rightarrow$  RLSimAnneal is
  mediator RLSAProgram
  domain-to-mediator  $\{F \rightarrow SimAnnealMain\}$       codomain-to-mediator  $\{\}$ 

ip-scheme-morphism Feas-to-RLSA : FeasibleSolution  $\rightarrow$  RLSimAnnealing is
  domain-sm  $\{\}$       mediator-sm  $\{\mathcal{R} \rightarrow Real\}$       codomain-sm  $\{\mathcal{R} \rightarrow Real\}$ 

```

Figure 101. Refinement of Abstract Simulated Annealing to Run-Length-Based

The mediator could be reorganized to make explicit the runs that are implicit in *SimAnneal-Step*, comparable to how single-trial hill climbing was optimized. In this case, however, the benefits of doing so seem minor.

Additional refinements of simulated annealing, either in the abstract or run-based, include

1. *GetNeighbor* can be refined to use a random number generator to select a candidate move. If one is careful, one can also generate candidates systematically without inadvertently biasing the search (14). This way an entire neighborhood is searched more efficiently and might form the basis for modifying the stopping criteria.
2. The initial temperature can be determined algorithmically by techniques such as starting at a very high temperature and doing some annealing in an abbreviated way to quickly lower it to a reasonable starting value. Similarly, a moderate or low initial temperature might be selected and then increased, as metal is annealed by first heating it quickly and then slowly cooling it.
3. Minimum and maximum run lengths can be made functions of the input size. The random number generator might be seeded by hashing the input in some fashion.
4. An alternative cooling schedule is to reduce the temperature at every trial, but by a very small amount. This schedule and the run-based schedule work about the same as long as they spend equivalent amounts of time over the same temperature ranges (14).

Many of these are independent as far as specifying them (that is, ignoring whether the combination makes sense or not algorithmically) and so can be organized as separate refinements that can be combined at will.

VII. *Advanced Local Search Techniques*

With the essential features of local search established and formalized in Chapter VI, we can now explore elaborations of local search that introduce new techniques for organizing and controlling the search to solve a problem more effectively or efficiently. The two techniques to be considered are concerned in particular with finding better local optima. Both feature a controlled use of worsening moves that enables the search to escape from one local optimum and find others. Both exploit problem structure not formalized by *LocalSearch* and so extend it with additional sorts and operations.

The first technique is *tabu search*, as described primarily by Fred Glover (23, 25, 24, 28, 27); the main ideas of tabu search were independently developed by Pierre Hansen under the name of Steepest Ascent Mildest Descent (35), and various others have refined the theory or reported applications of it to new problems. Those aspects of tabu search specifically directed to escaping from local optima will be analyzed in detail and formalized into an algebraic theory. The second technique is the Kernighan-Lin heuristic. Originally developed for the graph partitioning problem (44), it appears to have an unfilled potential for application to a wide variety of problems. The Kernighan-Lin heuristic requires the same theory extensions as tabu search, and the view developed here is that it is a specialized application of the more general tabu approach.

The results described in this chapter constitute a major research contribution. Neither of the techniques considered here has been the object of a formal treatment before this. Some of Glover's papers describe tabu search in abstract terms and employ mathematical notation in a descriptive role (23, 24, 28, 27), but not in a form rigorous or complete enough for software synthesis. Kernighan-Lin is rarely referenced outside of its application to the graph partitioning problem and seems to have been overlooked in the literature as an approach to local search problems in general.

7.1 *Tabu Search*

Beginning in the 1980's and continuing today, Fred Glover and others have introduced and elaborated new approaches to local search that aim to combine elements of artificial intelligence with traditional search techniques to make search more adaptive and effective. These new approaches are collectively known as tabu search. Eschewing stochastic approaches of all kinds, tabu search is characterized by the use of "memory structures" that guide search in various "strategic" ways. The classic example of such a memory structure is the *tabu list*. In the simplest case, a tabu list is a list of moves that are forbidden from being chosen. The basic selection rule is always to make the best move, even if it is to a worse solution. Unless care is taken, however, once a local optimum is found the search will take one step away from it and then immediately return. The tabu list prevents this. Each time a move is made, one or more moves are added to the tabu list in order to prevent the solution just abandoned from being visited again. The tabu list acts as a FIFO queue, dropping moves from tabu status after a certain number of other moves have been made. The length of the list defines how long a move will be held tabu, which is called its *tenure*. A tabu list is thus an example of a short-term, recency-based memory structure. A further refinement is the use of *aspiration criteria*, which provide a means for the tabu status of a move to be overridden when it can be proved that the solution reached is new.

While the tabu list is the characteristic feature of tabu search, the term has come to encompass myriad other memory-based techniques and approaches. Some are aimed at improving effectiveness, often by intensifying a search in a promising area of the solution space, or conversely by diversifying a search that is no longer producing good results by forcing it into an unexplored region. Intermediate and long-term memory structures based on frequency of move or solution attributes, for example, have been used to influence the selection of initial solutions in future runs (67), or used to influence the choice of individual moves, as in *shifting penalty* approaches (28). Sample problem data have been subjected to extremely intense searches so that the best local optima can

be analyzed for commonalities that can be incorporated into the algorithm as additional selection rules, a process called *target analysis* (25). Others techniques are aimed at efficiency. *Candidate list* strategies are used to identify a subset of a neighborhood that is likely to contain good neighbors, which saves time over searching the entire neighborhood (26, 38). Parallel implementations of tabu search have also been proposed (28). Most of these techniques bear little relationship to tabu lists and indeed have begun to find application in combination with other search techniques, such as genetic algorithms and simulated annealing (27) and even global search methods (20). Candidate lists and parallel processing are widely used throughout the optimization field and beyond. The only features of tabu search that are considered here are those directly related to the use of tabu lists, including aspiration criteria.

7.1.1 Principles of Tabu Search. Tabu lists can be used to control a search in many different ways, but the most fundamental is cycle prevention. In the strict sense, cycle prevention means that the search will never visit a solution more than once. Strict hill climbing clearly manages this because each solution is strictly better than the one before it. More generally, cycle prevention means that a *search state* never recurs, where the state consists not only of the current solution but also whatever other information is maintained and used to influence the choice of the next solution. Simulated annealing may revisit solutions, but its use of randomization makes it highly unlikely that it will follow more than once a trajectory of significant length through the search space. Here the search state includes the current seed value of the random number generator. Tabu search uses explicit rules to prevent cycling. Some rules prevent repetition of solutions and some only the weaker repetition of search states. In the context of tabu search, the search state is taken to be the current solution and the move to a new solution that is implied by the basic selection rule, which is that a move to a best neighbor that is not tabu will always be made, with ties broken in some as yet unspecified way.

The brute force approach to cycle prevention is to maintain a list of solutions that have been visited and to check each proposed move against this list. This can be very expensive to implement if solutions are large data structures or if comparing them takes a lot of time. It can also lead to inefficient search in that the search can visit a lot of very similar solutions, each distinct but all within a small region of the search space, when what may be needed is more substantial change to move far enough from one local optimum to find another. For reasons of computational and search efficiency, then, the brute force approach is not a good one.

An alternative is to record selected attributes of past solutions and/or moves. Attributes can be substantially smaller than entire solutions and quick to test for. They also introduce a degree of abstraction in that an attribute may be shared by many moves or many solutions. If a set of moves or solutions with some common feature is forbidden, then this injects a measure of diversification into the search. The dangers of overly general attributes are that they may include solutions that are actually good to visit, and they may restrict the search to the point that progress is seriously impeded. Aspiration criteria are used to combat precisely these weaknesses. The brute force approach has no need of aspiration because it excludes exactly the solutions previously visited and no others.

In the literature on tabu search, the term *move* is often used to refer to a transform variable without regard to the solution to which it is applied. It can also refer to a solution and transform variable pair, what Glover calls a solution-specific move (23). Since *N_{is}* is in general characterized as a relation rather than a function, the solution generated is part of a move. Finally, the input value clearly plays a vital role in determining which solutions and moves are valid. We will adopt the convention that elements of *C* will continue to be called transform variables and that *move* refers to a 4-tuple consisting of an input, two solutions and a transform value that transforms one into the other. Attributes of a move are sometimes classified as *from* attributes or *to* attributes when they more specifically refer to the solution before the move is made or after, respectively.

To support the general principles of tabu search, *Neighborhood* is extended with the following elements:

```

sort Move
sort-axiom Move = (D, R, C, R) | LegalMove
op LegalMove : D, R, C, R → Boolean
definition of LegalMove is
  axiom  $\forall (x : D, z : R, c : C, z' : R)$ 
     $(LegalMove(x, z, c, z') \Leftrightarrow I(x) \wedge O(x, z) \wedge N\_info(x, z, c) \wedge N\_is(x, z, c, z'))$ 
end-definition

sort T
op GetTabu : Move → T
op HasTabu : Move, T → Boolean

```

The sort *Move* packages all the information needed to describe a legal move. *T* is the sort of the attributes to be stored on the tabu list. The operation *GetTabu* extracts an attribute associated with a move and the operation *HasTabu* determines whether a move possesses a particular attribute. These extensions are independent of the cost function of *LocalSearch* and so are localized to *Neighborhood*. Since *Move* and *LegalMove* are already defined, a particular application of tabu search need only define *T*, *GetTabu* and *HasTabu*.

Axioms imposed on these operations define a *tabu strategy*. To prevent search states from recurring, for example, the following condition must hold:

$$\begin{aligned}
 &\forall (x : D, z : R, c : C, z' : R, z'' : R, c'' : C, z''' : R, m : Move, n : Move) \\
 &((x, z, c, z') = (\mathbf{relax} \text{ LegalMove})(m) \wedge N^*(x, z', z'')) \\
 &\wedge (x, z'', c'', z''') = (\mathbf{relax} \text{ LegalMove})(n) \\
 &\Rightarrow (m = n \Rightarrow HasTabu(n, GetTabu(m)))
 \end{aligned}$$

This says that possession of the move attribute identified by *GetTabu* is a necessary condition of a search state recurring, so that by forbidding moves that satisfy *HasTabu*, recurrence is prevented. The term $N^*(x, z', z'')$ represents an arbitrarily long sequence of moves. If the neighborhood is reachable, z'' could be any feasible solution whatever. This condition is a very broad characteriza-

tion of what tabu rules are to accomplish and there are many ways to satisfy it. Specific approaches typically focus on just one aspect of the search state.

One specialization focuses on the transform variables used. Glover describes a strategy that in our formalism defines

```

sort-axiom  $T = C$ 
op  $GetTabu : D, R, C, R \rightarrow T$ 
definition of  $GetTabu$  is
    axiom  $GetTabu = \lambda(x : D, z : R, c : C, z' : R) c$ 
end-definition

op  $HasTabu : D, R, C, R, T \rightarrow Boolean$ 
definition of  $HasTabu$  is
    axiom  $HasTabu = \lambda(x : D, z : R, c : C, z' : R, t : T) (t = c)$ 
end-definition

```

The signatures for *GetTabu* and *HasTabu* have been relaxed here because the definitions for them work on illegal moves as well as legal ones. Below an interpretation will be given that refines the operations on *Move* given above to this relaxed form. The strategy represented is that once a move applying a particular transform variable c has been made, all moves involving c are forbidden regardless of the solution to which c would be applied (23). This is called a *repeat move* strategy because it classifies a move as tabu whenever it might repeat a move made previously. Clearly this strategy is strong enough to prevent recurrence of a search state. It is a simple rule, and applicable to any problem or neighborhood structure uniformly. It does not prevent recurrence of a solution, however, and in practice it does not work well. One can see that a locally optimal solution would tend to be revisited as many times as there are unique trajectories leading back to it before this rule would force the search to move on. This rule also has the potential to forbid many moves that do not lead to recurrence and so restrict the search unnecessarily. Perhaps because of these inherent weaknesses, repeat move strategies that include move attributes other than or in addition to the transform variable have not been reported in the literature.

Strict cycle prevention focuses on preventing z from being revisited in the first place, rather than on which move will be taken if it is revisited. Strict cycle prevention is described by the

condition

$$\begin{aligned}
& \forall (x : D, z : R, c : C, z' : R, z'' : R, c'' : C, z''' : R, m : Move, n : Move) \\
& (\langle x, z, c, z' \rangle = (\mathbf{relax} \text{ LegalMove})(m) \wedge N^*(x, z', z'')) \\
& \wedge \langle x, z'', c'', z''' \rangle = (\mathbf{relax} \text{ LegalMove})(n) \\
& \Rightarrow (z = z''' \Rightarrow HasTabu(n, GetTabu(m)))
\end{aligned}$$

That is, an attribute t is associated with a move from z to z' such that any future move back to z will possess that attribute; preventing moves with that attribute is sufficient to prevent z from being revisited. As before, the converse is not true: just because a move possesses a tabu attribute, it need not be a move back to z . This strategy is more powerful than repeat move because it forces the search to stay away from z . When improving moves exist, it is the natural tendency of the basic selection rule to generate new solutions. When trying to escape from a local optimum, the natural tendency is to move away and then return. This rule opposes that tendency.

In his earlier work, Glover recommends using a heuristic approximation to this condition (which he only describes informally) that is based on *inverse moves* (23). An inverse move is one that undoes the effect of a move made previously; in particular, it is the move that immediately returns the search to the solution just left. Inverse moves are only defined for symmetric neighborhoods. A formal characterization of this strategy is

```

sort-axiom  $T = C$ 
op  $GetTabu : Move \rightarrow T$ 
definition of  $GetTabu$  is
  axiom  $\forall (m : Move, x : D, z : R, c : C, z' : R)$ 
     $(\langle x, z, c, z' \rangle = (\mathbf{relax} \text{ LegalMove})(m)$ 
       $\Rightarrow N\_info(x, z', GetTabu(m)) \wedge N\_is(x, z', GetTabu(m), z))$ 
end-definition
op  $HasTabu : D, R, C, R, T \rightarrow Boolean$ 
definition of  $HasTabu$  is
  axiom  $HasTabu = \lambda(x : D, z : R, c : C, z' : R, t : T) (t = c)$ 
end-definition

```

A relaxed signature is used again for *HasTabu*, but not for *GetTabu* since only legal moves have inverses. As long as the neighborhood is symmetric, a transform value exists that satisfies the axiom for *GetTabu*. As long as the neighborhood is unique, the inverse is unique and the axiom for *GetTabu* is a definition.

The inverse move strategy does *not* necessarily satisfy the strict cycle prevention condition. In the k -subset-1-exchange neighborhood, for example, a move made by applying the transform $\langle i, j \rangle$ is immediately undone by applying the transform $\langle j, i \rangle$. If $k \geq 3$ and we represent elements of S by letters, then the sequence of moves

$$\langle a, b \rangle \langle c, a \rangle \langle b, c \rangle$$

returns all elements to their original positions (either in or out of the current subset) without ever executing the inverse of any move made. It takes more than one move to do this, however, and the inverse move strategy is successful when sufficient moves are required that the search has escaped one local optimum and begun approaching another. When used in conjunction with the repeat move strategy, search state recurrence is prevented and the efficiency of the search is greatly enhanced. Since the repeat move and inverse move strategies use the same definition of T and *HasTabu* and similar axioms for *GetTabu*, they are particularly well suited to combined application.

Two other strategies are implicitly discussed (by example, not as strategies) in (27, 28). We term these *lock in* and *lock out*. Both are based on attributes of solutions as well as transform variables, and both are sufficient to prevent solutions from recurring. Unfortunately, both are informal strategies: while it has been easy to apply them to a wide variety of neighborhoods and prove that they work, no formal definitions like those given above that describe in general how they work are available.

The lock out strategy makes use of from-attributes to prevent solutions from recurring. The goal is to identify attributes that are affected by N_{is} and to store these in the tabu list. Future

moves are tabu if they produce solutions possessing all of these attributes: the attributes are *locked out*, or prevented from occurring together. For k -subset-1-exchange, for example, the move $\langle i, j \rangle$ moves element i into the current subset and moves j out. The lock out strategy would forbid future subsets where j was in the subset and i was out. As long as at least one of these attributes is not satisfied, the new solution cannot be the old from-solution. Formally we define

```

sort  $KS-T$ 
sort-axiom  $KS-T = KS-C$ 

op  $KS-GetTabu : KS-D, Set, KS-C, Set \rightarrow KS-T$ 
definition of  $KS-GetTabu$  is
  axiom  $KS-GetTabu = \lambda(x : KS-D, A : Set, c : KS-C, A' : Set) c$ 
end-definition

op  $KS-HasTabu : KS-D, Set, KS-C, Set, KS-T \rightarrow Boolean$ 
definition of  $KS-HasTabu$  is
  axiom  $\forall(x : KS-D, A : Set, c : KS-C, A' : Set, i : E, j : E)$ 
     $(HasTabu(x, A, c, A', \langle i, j \rangle) \Leftrightarrow in(j, A') \wedge \neg in(i, A'))$ 
end-definition

```

The sort T in this case again has the same form as C , but the *HasTabu* test makes it clear that T means something quite different. The move or moves forbidden by a given element of sort T depend on the solution produced by applying that move to a particular solution, not merely on the transform variables involved. The *HasTabu* test does not even consider the transform variable of the move, but only the to-solution.

A comparison with the inverse move strategy for this neighborhood is instructive. After move $\langle a, b \rangle$, both strategies forbid the move applying $\langle b, a \rangle$ and neither forbids a move such as applying $\langle c, a \rangle$. If this move is made, both strategies add $\langle a, c \rangle$ to the forbidden list. Here a difference becomes apparent. For the inverse move strategy, the earlier element $\langle b, a \rangle$ is essentially inactive because it represents a transform that is not applicable to the current solution since a is not in the current solution. For the lock-out strategy, however, it represents a situation that would occur if the transform $\langle b, c \rangle$ were now applied. Such a move is thus tabu for lock-out, avoiding the recurrence that inverse move allows.

The lock in strategy is dual to lock out, making use of to-attributes. Here the goal again is to identify attributes affected by *N_is*, but this time it is the to-solution whose attributes are stored. Future moves are tabu if they produce solutions that do *not* possess all of these attributes: the attributes are *locked in*, or prevented from changing. For *k*-subset-1-exchange, the lock in strategy applied to the move $\langle i, j \rangle$ requires all future solutions to have *i* in the subset and *j* not. Neither can be moved. The inverse move $\langle j, i \rangle$ is tabu, as are all moves of the form $\langle x, i \rangle$ or $\langle j, y \rangle$, where *x* and *y* are arbitrary elements of *S*. Formally we define

```

sort KS-T
sort-axiom KS-T = KS-C

op KS-GetTabu : KS-D, Set, KS-C, Set → KS-T
definition of KS-GetTabu is
  axiom KS-GetTabu =  $\lambda(x : KS-D, A : Set, c : KS-C, A' : Set) c$ 
end-definition

op KS-HasTabu : KS-D, Set, KS-C, Set, KS-T → Boolean
definition of KS-HasTabu is
  axiom  $\forall(x : KS-D, A : Set, c : KS-C, A' : Set, i : E, j : E)$ 
     $(HasTabu(x, A, c, A', \langle i, j \rangle) \Leftrightarrow \neg(in(i, A') \wedge \neg in(j, A')))$ 
end-definition

```

Notice in this case that *HasTabu* can be rewritten by DeMorgan's laws as

$$in(j, A') \vee \neg in(i, A')$$

which has the same two terms as the lock-out strategy but is a disjunction rather than a conjunction. Lock-in is a much weaker, that is more inclusive, condition and thus restricts the search more strongly. It forces the search to continue in a particular direction and thus to leave a local optimum quickly. It also carries a correspondingly greater risk of being too restrictive.

An alternative definition of *KS-HasTabu* for lock-in that has the form of a test on transform variables rather than the to-solution is

```

definition of KS-HasTabu is
  axiom  $\forall(x : KS-D, A : Set, k : E, l : E, A' : Set, i : E, j : E)$ 
     $(HasTabu(x, A, \langle k, l \rangle, A', \langle i, j \rangle) \Leftrightarrow k = j \vee l = i)$ 
end-definition

```

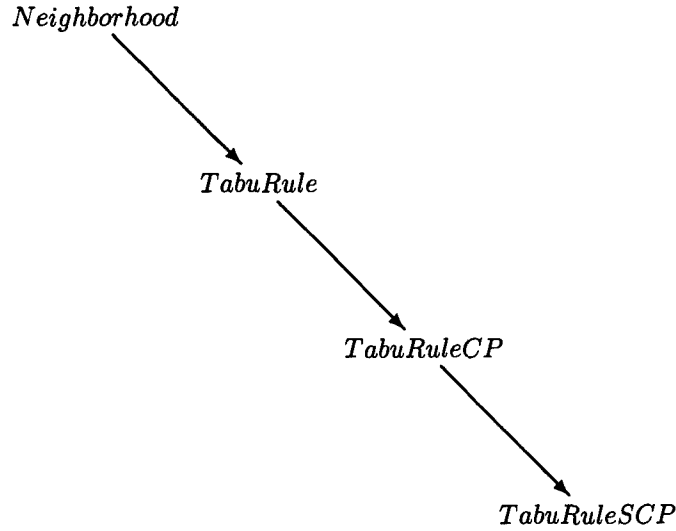

This works because the only way for an element to move is for a transform variable to move it. This reformulation will probably be more efficient computationally, since it compares elements rather than computing set membership, but the relationship between lock in and lock out is more clearly shown by the original definition. Design optimization is a valid but separate concern. Not all lock in strategies have such optimizations, though many do.

Figures 102 through 105 summarize the discussion of tabu strategies. Figure 102 shows how the spec *Neighborhood* is refined successively by *TabuRule*, which adds the basic elements needed, and additional specs that refine it first to weak or state cycle prevention and then to strong or solution cycle prevention. These specs represent *abstract tabu strategies* that are not realized directly for particular neighborhoods. Figures 103 through 105 show *concrete tabu strategies* that can be applied systematically to neighborhood specifications. When a strategy is instantiated for a particular neighborhood, it defines a *tabu rule*, hence the name of the first spec.

Figure 103 shows the inverse move strategy. It refines only *TabuRule* because it makes no guarantees about its cycle prevention abilities. An interpretation is used in order to relax the signature for *HasTabu* used in *TabuRule*. Both the source and target specs of this interpretation have morphisms from *SymmetricNeighborhood*, a fact that will be exploited when the strategy is instantiated for a particular neighborhood.

The repeat move strategy, with specs in Figure 104, refines *TabuRuleCP* because it is strong enough to guarantee weak cycle prevention. Repeat move and inverse move are combined in *TabuRuleRIM*, with specs in Figure 105. Like inverse move, each of these is presented as an interpretation to reconcile the different signatures used, and again both the sources and the targets of these interpretations have morphisms from *Neighborhood*.

A design optimization that could be made to the RM strategy is to notice that *RMGetTabu* and *RMHasTabu* have the same definition and so to delete one copy and define both *GetTabu* and *HasTabu* in the mediator in terms of the remaining one. This carries with it a small risk, in that



```

spec TabuRule is
  import Neighborhood

  sort Move
  sort-axiom Move = (D, R, C, R) | LegalMove

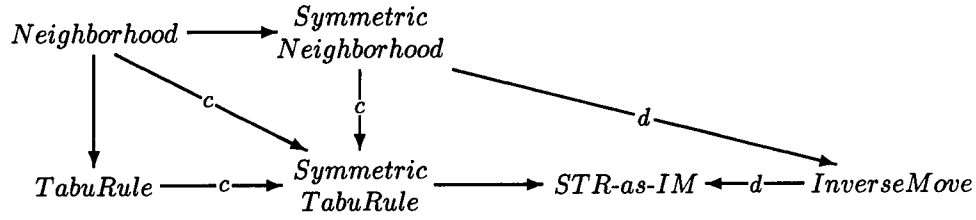
  op LegalMove : D, R, C, R → Boolean
  definition of LegalMove is
    axiom  $\forall (x : D, z : R, c : C, z' : R)$ 
       $(LegalMove(x, z, c, z') \Leftrightarrow I(x) \wedge O(x, z) \wedge N\_info(x, z, c) \wedge N\_is(x, z, c, z'))$ 
  end-definition

  sort T
  op GetTabu : Move → T
  op HasTabu : Move, T → Boolean
end-spec

spec TabuRuleCP is
  import TabuRule
  axiom  $\forall (x : D, z : R, c : C, z' : R, z'' : R, c'' : C, z''' : R, m : Move, n : Move)$ 
     $(\langle x, z, c, z' \rangle = (\mathbf{relax} \text{ } LegalMove)(m) \wedge N^*(x, z', z''))$ 
     $\wedge \langle x, z'', c'', z''' \rangle = (\mathbf{relax} \text{ } LegalMove)(n)$ 
     $\Rightarrow (m = n \Rightarrow HasTabu(n, GetTabu(m)))$ 
end-spec

spec TabuRuleSCP is
  import TabuRule
  axiom  $\forall (x : D, z : R, c : C, z' : R, z'' : R, c'' : C, z''' : R, m : Move, n : Move)$ 
     $(\langle x, z, c, z' \rangle = (\mathbf{relax} \text{ } LegalMove)(m) \wedge N^*(x, z', z''))$ 
     $\wedge \langle x, z'', c'', z''' \rangle = (\mathbf{relax} \text{ } LegalMove)(n)$ 
     $\Rightarrow (z = z''' \Rightarrow HasTabu(n, GetTabu(m)))$ 
end-spec
  
```

Figure 102. Hierarchy of Specs for Tabu Rules



```

spec SymmetricTabuRule is
  colimit of diagram
    nodes Neighborhood, TabuRule, SymmetricNeighborhood
    arcs Neighborhood → TabuRule : import-morphism,
         Neighborhood → SymmetricNeighborhood : import-morphism
  end-diagram

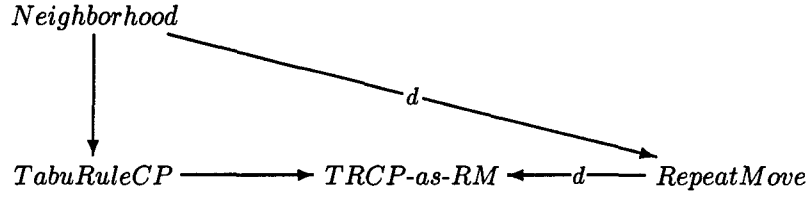
spec InverseMove is
  import SymmetricNeighborhood
  sort Move
  sort-axiom Move = (D, R, C, R) | LegalMove
  op LegalMove : D, R, C, R → Boolean
  definition of LegalMove is
    axiom  $\forall(x : D, z : R, c : C, z' : R)$ 
       $(LegalMove(x, z, c, z') \Leftrightarrow I(x) \wedge O(x, z) \wedge N\_info(x, z, c) \wedge N\_is(x, z, c, z'))$ 
  end-definition
  op IMGetTabu : Move → C
  definition of IMGetTabu is
    axiom  $\forall(m : Move, x : D, z : R, c : C, z' : R)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m) \Rightarrow N\_info(x, z', GetTabu(m)) \wedge N\_is(x, z', GetTabu(m), z))$ 
  end-definition
  op IMHasTabu1 : D, R, C, R, C → Boolean
  definition of IMHasTabu1 is
    axiom  $IMHasTabu1 = \lambda(x : D, z : R, c : C, z' : R, t : C) (t = c)$ 
  end-definition
end-spec

spec STR-as-IM is
  import InverseMove
  op IMHasTabu : Move, C → Boolean
  definition of HasTabu is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move) (\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m) \Rightarrow (HasTabu(m, t) \Leftrightarrow IMHasTabu1(x, z, c, z', t)))$ 
  end-definition
end-spec

interpretation STR-to-IM : SymmetricTabuRule  $\Rightarrow$  InverseMove is
  mediator STR-as-IM
  domain-to-mediator {T → C, GetTabu → IMGetTabu, HasTabu → IMHasTabu}
  codomain-to-mediator import-morphism

```

Figure 103. Inverse Move Tabu Strategy



```

spec RepeatMove is
  import Neighborhood

  op RMGetTabu1 : D, R, C, R → C
  definition of RMGetTabu1 is
    axiom RMGetTabu1 = λ(x : D, z : R, c : C, z' : R) c
  end-definition

  op RMHasTabu1 : D, R, C, R, C → Boolean
  definition of RMHasTabu1 is
    axiom RMHasTabu1 = λ(x : D, z : R, c : C, z' : R, t : C) (t = c)
  end-definition
end-spec

spec TRCP-as-RM is
  import RepeatMove

  sort Move
  sort-axiom Move = (D, R, C, R) | LegalMove

  op LegalMove : D, R, C, R → Boolean
  definition of LegalMove is
    axiom ∀(x : D, z : R, c : C, z' : R)
      (LegalMove(x, z, c, z') ⇔ I(x) ∧ O(x, z) ∧ N.info(x, z, c) ∧ N.is(x, z, c, z'))
  end-definition

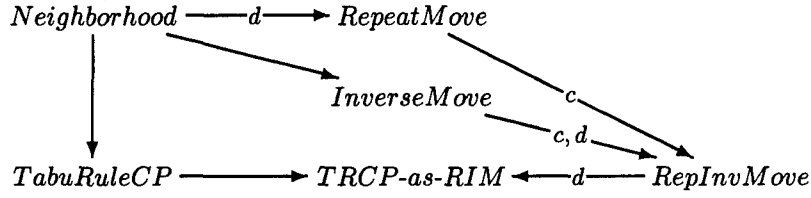
  op RMGetTabu : Move → C
  definition of RMGetTabu is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move) ((x, z, c, z') = (relax LegalMove)(m)
      ⇒ RMGetTabu(m) = RMGetTabu1(x, z, c, z'))
  end-definition

  op RMHasTabu : Move, C → Boolean
  definition of RMHasTabu is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move, t : C) ((x, z, c, z') = (relax LegalMove)(m)
      ⇒ (RMHasTabu(m, t) ⇔ RMHasTabu1(x, z, c, z', t)))
  end-definition
end-spec

interpretation TRCP-to-RM : TabuRuleCP ⇒ RepeatMove is
  mediator TRCP-as-RM
  domain-to-mediator {T → C, GetTabu → RMGetTabu, HasTabu → RMHasTabu}
  codomain-to-mediator import-morphism

```

Figure 104. Repeat Move Tabu Strategy



```

spec RepInvMove is
  colimit of diagram
    nodes Neighborhood, InverseMove, RepeatMove
    arcs   Neighborhood → InverseMove : {}
           Neighborhood → RepeatMove : import-morphism
  end-diagram

spec TRCP-as-RIM is
  import RepInvMove

  sort Move
  sort-axiom Move = (D, R, C, R) | LegalMove

  op LegalMove : D, R, C, R → Boolean
  definition of LegalMove is
    axiom ∀(x : D, z : R, c : C, z' : R)
      (LegalMove(x, z, c, z') ⇔ I(x) ∧ O(x, z) ∧ N_info(x, z, c) ∧ N_is(x, z, c, z'))
  end-definition

  sort T
  sort-axiom T = C, C

  op RIMGetTabu : Move → T
  definition of RIMGetTabu is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move)
      ((x, z, c, z') = (relax LegalMove)(m)
        ⇒ RIMGetTabu(m) = (IMGetTabu(m), RMGetTabu1(x, z, c, z')))
  end-definition

  op RIMHasTabu : Move, T → Boolean
  definition of RIMHasTabu is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move, t : C, t' : C)
      ((x, z, c, z') = (relax LegalMove)(m)
        ⇒ (RIMHasTabu(m, ⟨t, t'⟩)
          ⇔ IMHasTabu1(x, z, c, z', t) ∨ RMHasTabu1(x, z, c, z', t')))
  end-definition
end-spec

interpretation TRCP-to-RIM : TabuRuleCP ⇒ RepInvMove is
  mediator TRCP-as-RIM
  domain-to-mediator {GetTabu → RIMGetTabu, HasTabu → RIMHasTabu}
  codomain-to-mediator import-morphism

```

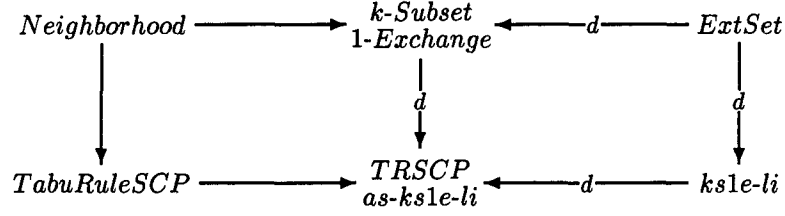
Figure 105. Repeat and Inverse Move Combined Tabu Strategy

once the two operations have been identified, they cannot be separated. If the difference in how they are used eventually motivates a desire to implement or refine them differently, that would no longer be an option. The designer would have to include this factor in the decision to make the optimization or not.

The lock in and lock out strategies are not formalized as concrete strategies and so do not appear as specializations of *TabuRuleSCP*. They are present only implicitly, as a library of refinements for particular neighborhoods.

Tabu strategies are applied to particular neighborhoods by means of an interpretation morphism from a neighborhood specification, which is a refinement of *Neighborhood*, to a tabu rule specification, which is a refinement of *TabuRule*. A complete specification of the lock-in strategy applied to *k*-subset-1-exchange is shown in Figures 106 and 107 as an example. Application of a strategy may require the domain theory for the neighborhood to be extended by more than just definitions, though in this case it did not. Two levels of definition are used to provide operations with relaxed signatures for later refinement yet reconcile them with the stricter signatures in *TabuRule*. This is not required for correctness but is a minor optimization that removes some unneeded complexity, namely the use of subsorts.

Instantiating a concrete strategy such as inverse move is a syntactic operation that could easily be automated. Figure 108 shows how the inverse move strategy is instantiated for the *k*-subset-1-exchange neighborhood. This neighborhood is a refinement of *PerfectSymmetricNeighborhood*, so it is also a refinement of *SymmetricNeighborhood* (via an interpretation named *S-ks1e*, say). *InverseMove* is a definitional extension of *SymmetricNeighborhood*, so *S-ks1e* is uniquely extended to it by pushout. This interpretation is composed with *STR-to-IM* to give the instantiated tabu rule *ks1e-im*. *InverseMove* contains a non-constructive definition of *IMGetTabu*, so after instantiation this must be reformulated to compute inverse moves for the particular neighborhood in question, for example by unskolemization.



```

spec ksle-li is
  import k-Subset-1-Exchange

  op KS-GetTabu1 : KS-D, Set, KS-C, Set → KS-C
  definition of KS-GetTabu1 is
    axiom KS-GetTabu1 =  $\lambda(x : KS-D, A : Set, c : KS-C, A' : Set) c$ 
  end-definition

  op KS-HasTabu1 : KS-D, Set, KS-C, Set, KS-C → Boolean
  definition of KS-HasTabu1 is
    axiom  $\forall(x : KS-D, A : Set, c : KS-C, A' : Set, i : E, j : E)$ 
       $(KS-HasTabu1(x, A, c, A', \langle i, j \rangle) \Leftrightarrow in(j, A') \vee \neg in(i, A'))$ 
  end-definition
end-spec

spec TRSCP-as-ksle-li is
  import ksle-li

  sort KS-Move
  sort-axiom KS-Move = (D, R, C, R) | KS-LegalMove

  op KS-LegalMove : D, R, C, R → Boolean
  definition of KS-LegalMove is
    axiom  $\forall(x : D, z : R, c : C, z' : R)$ 
       $(KS-LegalMove(x, z, c, z') \Leftrightarrow$ 
         $KS-I(x) \wedge KS-O(x, z) \wedge Exchange\_info(x, z, c) \wedge Exchange(x, z, c, z'))$ 
  end-definition

  op KS-GetTabu : Move → C
  definition of KS-GetTabu is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } KS-LegalMove)(m)$ 
         $\Rightarrow KS-GetTabu(m) = KS-GetTabu1(x, z, c, z'))$ 
  end-definition

  op KS-HasTabu : Move, C → Boolean
  definition of KS-HasTabu is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move, t : C)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } KS-LegalMove)(m)$ 
         $\Rightarrow (KS-HasTabu(m, t) \Leftrightarrow KS-HasTabu1(x, z, c, z', t)))$ 
  end-definition
end-spec

```

Figure 106. Lock-In Strategy Applied to k -Subset-1-Exchange Neighborhood

interpretation $ks1e-li : TabuRuleSCP \Rightarrow ks1e-li$ is
mediator $TRSCP-as-ks1e-li$
domain-to-mediator
 $\{D \rightarrow KS-D, I \rightarrow KS-I, R \rightarrow Set, O \rightarrow KS-O, C \rightarrow KS-C,$
 $N\_info \rightarrow Exchange\_info, N\_is \rightarrow Exchange, N \rightarrow KS-N, N^* \rightarrow KS-N^*,$
 $T \rightarrow KS-C, Move \rightarrow KS-Move, LegalMove \rightarrow KS-LegalMove,$
 $GetTabu \rightarrow KS-GetTabu, HasTabu \rightarrow KS-HasTabu\}$
codomain-to-mediator import-morphism

ip-scheme-morphism $ks1e-li : ks1e \rightarrow ks1e-li$ is
domain-sm $\{\}$ **mediator-sm** $\{\}$ **codomain-sm** $\{\}$

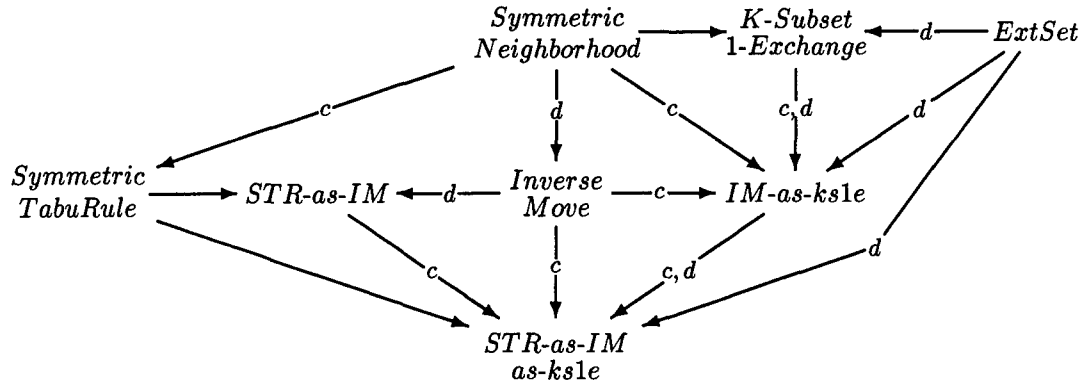
Figure 107. Lock-In Strategy Applied to k -Subset-1-Exchange Neighborhood, Cont.

All of these strategies and rules, abstract and concrete, parameterized and instantiated, are refinements of *Neighborhood* and are incorporated into *LocalSearch* in the obvious way. Figure 109 shows a spec *TabuSearch* that incorporates the elements of *TabuRule*.

7.1.2 Variations and Examples of Tabu Strategy. Like any self-respecting algorithmic technique, controlling search by means of tabu lists has a number of significant variations and its application to specific cases often brings out distinctive features worthy of note. This section enumerates some of the possibilities. Specifications for neighborhoods mentioned here are found in Appendix A. Tabu rules for each neighborhood in the library are discussed there, with particularly interesting cases described in more detail below.

7.1.2.1 Multi-list Tabu Strategies. If one tabu list is good, several lists might be better. Separate attributes can be used to control different aspects of search separately. This entails defining for each list its own T sort and associated *GetTabu* and *HasTabu* operations, and implementing in the program scheme a list management strategy for coordinating multiple lists. Several strategies are possible, each with its own characteristic features and benefits.

Multiple lists can be used in such a way that a move is tabu whenever it is classified tabu by any of the lists individually. The effect is largely the same as using one big list containing tuples and a test that evaluates each element of the tuple disjunctively, as was pointed out above



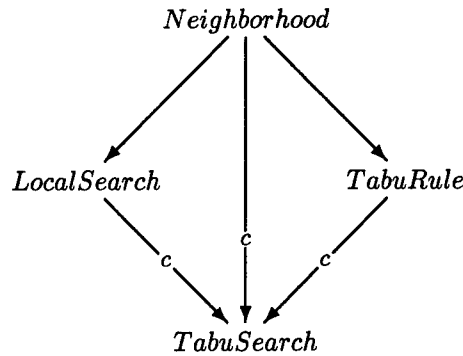
spec *IM-as-ksle* is
 colimit of diagram
 nodes *SymmetricNeighborhood*, *k-Subset-1-Exchange*, *InverseMove*
 arcs *SymmetricNeighborhood* → *k-Subset-1-Exchange* :
 $\{D \rightarrow KS-D, I \rightarrow KS-I, R \rightarrow Set, O \rightarrow KS-O, C \rightarrow KS-C,$
 $N\_info \rightarrow Exchange\_info, N\_is \rightarrow Exchange, N \rightarrow KS-N, N^* \rightarrow KS-N^*\},$
SymmetricNeighborhood → *InverseMove* : **import-morphism**
 end-diagram

spec *STR-as-IM-as-ksle* is
 colimit of diagram
 nodes *InverseMove*, *IM-as-ksle*, *STR-as-IM*
 arcs *InverseMove* → *IM-as-ksle* : **cocone-morphism from InverseMove**,
InverseMove → *STR-as-IM* : **import-morphism**
 end-diagram

interpretation *ksle-im* : *SymmetricTabuRule* ⇒ *ExtSet* is
 mediator *STR-as-IM-as-ksle*
 domain-to-mediator $\{D \rightarrow KS-D, I \rightarrow KS-I, R \rightarrow Set, O \rightarrow KS-O, C \rightarrow KS-C,$
 $N\_info \rightarrow Exchange\_info, N\_is \rightarrow Exchange, N \rightarrow KS-N, N^* \rightarrow KS-N^*,$
 $T \rightarrow C, GetTabu \rightarrow IMGetTabu, HasTabu \rightarrow IMHasTabu\}$
 codomain-to-mediator $\{\}$

ip-scheme-morphism *ksle-im* : *S-ksle* → *ksle-im* is
 domain-sm **cocone-morphism from *SymmetricNeighborhood***
 mediator-sm $\{\}$
 codomain-sm **identity-morphism**

Figure 108. Inverse Move Strategy Instantiated for *k*-Subset-1-Exchange Neighborhood



spec *TabuSearch* is
colimit of diagram
nodes *Neighborhood, LocalSearch, TabuRule*
arcs *Neighborhood* $\rightarrow$ *LocalSearch* : **cocone-morphism from** *Neighborhood*,
Neighborhood $\rightarrow$ *TabuRule* : **import-morphism**
end-diagram

Figure 109. Incorporating *TabuRule* into *LocalSearch*

in connection with the combined repeat and inverse move strategy. In either case just one match anywhere in a list or tuple is sufficient to classify a move as tabu. The difference is that the multiple list implementation permits separate attributes to have different tenures. That is, separate lists can be of different lengths, whereas the tuple approach constrains all attributes to have the same tenure by being grouped together on a single list. Tenure, or the length of a tabu list, is a major tunable parameter in program schemes for tabu search, and using different tenures for different attributes can have powerful effects. For the traveling salesman problem, for example, the k -subset-1-exchange neighborhood can be used to search among sets of edges representing tours. The size of a tour is much smaller than the total number of edges, so the impact of locking an edge into a tour is much greater than locking one out (or more precisely, locking an edge into not belonging to the current solution). The lock-in rule for k -subset-1-exchange treats both attributes equally, but search is enhanced if they are separated into different lists so that edges locked into the current solution can be given a shorter tenure than edges locked into the complement. For the graph partitioning

problem, on the other hand, the number of nodes in the current solution is the same as the number excluded, so equal treatment is appropriate.

Multiple lists can also be used in such a way that a move is tabu only if it is classified tabu by all of the lists. This can be done in two distinct ways. First, a move may be tabu only if it is classified tabu by *corresponding elements* of the lists. This is equivalent to implementing a single list where T is a tuple sort and the *HasTabu* test is a conjunction, as for the lock-out rule for k -subset-1-exchange above. It makes no sense to have lists of different lengths, because the excess elements of the longer lists have no partners to be conjuncted with on the other lists. The other way to conjunct multiple lists relaxes the correspondence condition. That is, a move must be classified tabu by each list, but the classifying elements on the lists need not have any particular relationship with each other. This is subtly different in its effect. When lists are combined disjunctively, a move is tabu as soon as it is classified tabu by one list element from any list and will not become available again until all such classifications are cleared. When lists are combined in correspondence, or when tuples are used, a move is tabu when it is classified so by a set of attributes occurring together. Other sets of attributes may classify some attributes of a move as tabu without having a complete match. That is each list, considered separately, may have multiple elements classifying some attributes of a move as tabu, yet the move as a whole may not be tabu at all. A move returns to non-tabu status as soon as any one attribute in each set no longer classifies it so. When lists are combined conjunctively but freely, an intermediate situation occurs. Attributes of a move can be classified tabu by multiple elements of all the lists but one, as long as the remaining list does not have any such elements. With both conjunctive strategies, the status of a move changes in subtle ways as elements are added to and dropped from the tabu lists, even if it never changes its tabu status *per se*. In summary, the free conjunction strategy is more restrictive than the correspondence strategy and less restrictive than the disjunctive strategy, though closer to the former. Moreover, the free strategy again permits separate lists to have distinct tenures.

List management strategies can become arbitrarily complex by combining the basic strategies that have been described. One set of attributes might be treated conjunctively on separate lists, but disjunctively as a group with respect to some other set of attributes. This complexity can be controlled somewhat by defining a normal form in analogy to conjunctive normal form for formulas in propositional calculus. Specs could be written to describe how sets of sets of lists are manipulated in general, and these specs could then be refined to describe specific sets or tuples of attributes for a particular problem. Such complexity is rare, however, and having specs for common, relatively simple cases stored in the knowledge base also has merit.

Multiple lists imply the use of multiple attributes. How are suitable attributes identified? One way is to combine tabu strategies, as repeat and inverse are often combined. Each contributes one or more attributes, and the user should consider whether combining them disjunctively or conjunctively makes more sense, or even experiment to see which one works better on his or her problem. Another way is to split apart a strategy or a rule. If the sort T is refined to a tuple by a strategy or an instantiation of a strategy for a particular neighborhood, or if the *GetTabu* or *HasTabu* tests are themselves conjunctions or disjunctions, this strongly suggests attributes that might be separated. Distinct attributes need not have distinct sorts or even distinct tests; they might logically be just different occurrences of the same attribute, as for example the various elements of a sequence are generally treated the same. Once separated, attributes might be recombined differently, changing a conjunction to a disjunction or vice versa in order to relax or tighten the search constraints and generate more options.

7.1.2.2 Tabu Rules for the k -Subset-1-Exchange Neighborhood. Many tabu rules for the k -subset-1-exchange neighborhood have already been described to illustrate the tabu strategies. Figure 110 shows relations among several rules for this neighborhood. Each rule is represented by a formula or condition where A' is the solution that will be produced if an unspecified candidate move is made, and i and j are elements from a tabu list. The strictest test, and therefore the

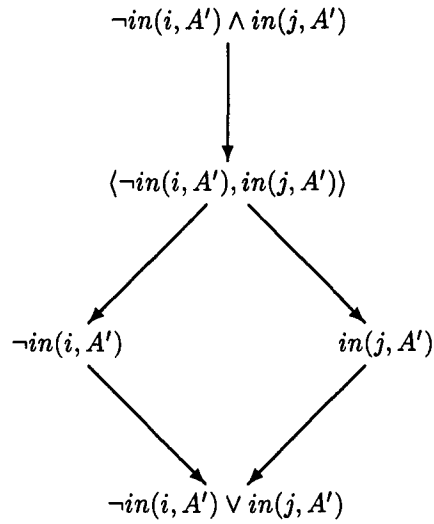


Figure 110. Tabu Rules for k -Subset-1-Exchange Derived from the Lock-Out Rule

least restrictive as a tabu rule, is at the top of the diagram and the arrows show how it can be successively relaxed in various ways, terminating in the weakest, and therefore most inclusive and restrictive test at the bottom. At the top is the *HasTabu* test for the lock-out rule given previously. The sort $T = KS-C$ is a tuple, and the test is a conjunction. The two set elements, the edge added and the edge dropped, can be treated as separate attributes. The next test down represents the attributes on separate lists combined with free conjunction. The two branches below this one are individual lists: the attribute not shown has been dropped completely from the tabu rule. That is, for the left branch a move is tabu if and only if it attempts to drop an edge previously added, and for the right branch a move is tabu if and only if it attempts to add an edge previously dropped. The bottom test has both attributes tested disjunctively. The diagram does not portray whether lists of different lengths are used or not for the tests where this is an option, though clearly this makes a significant difference in practice. Since the least restrictive rule is strong enough to prevent solutions from recurring, so are all the others. Preventing a dropped edge from being added back, for example, is by itself sufficient to ensure this.

The lock-in rule can be similarly decomposed, and in this case the same diagram is generated. The bottom test is equivalent to the lock-in rule, by DeMorgan's law. The two attributes can be separated. Either can be used in isolation, or they can be conjuncted freely or in coordination, resulting in the lock-out rule at the top of the diagram, again by DeMorgan's law.

The inverse move rule does not appear in Figure 110. This rule does not relate directly to any of the rules shown because it permits moves they forbid, and forbids moves they permit. The T sort used is again a tuple, however, and tuple equality is implicitly a conjunction of equalities on components, so separate attributes can be defined and used to generate the diagram shown in Figure 111. Here i and j are from the tabu list and i' and j' form a candidate move. The top test is the inverse move rule; we have already seen that it is not equivalent to lock-out. The other tests, however, are equivalent to tests in Figure 110. Elements of S can only be in A' or not in A' , and they only change position by appearing in a transform variable in particular positions. Thus testing whether a candidate move has i in the first position (i.e., adds i to the solution) is equivalent to testing whether the solution produced by this move contains i , and so on. The bottom test, therefore, is equivalent to the lock-in rule, and the single-attribute tests in both diagrams are of course equivalent as well. The free conjunction of the single-attribute tests is also the same as the corresponding test based on lock-out attributes. Thus only inverse move itself is a distinct rule.

Decomposing the repeat move rule yields yet another diagram, not shown. The element extracted from a move by *RMGetTabu* is the transform variable, so this rule forbids adding the edge just added and deleting the edge just deleted. The rules already considered, except inverse move proper, prevent such moves from being possible, since for example an added edge must be deleted before it can be added again. The repeat move rule is a conjunction, so it would appear at the top of the diagram as the least restrictive rule. The other rules are more restrictive, but still suffer from the inherent weakness of the strategy and so are not likely to be useful.

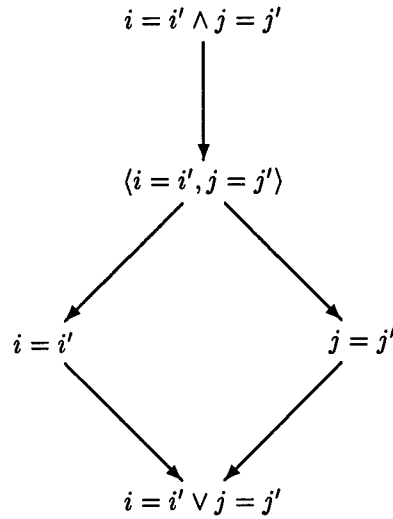


Figure 111. Tabu Rules for k -Subset-1-Exchange Derived from the Inverse Move Rule

7.1.2.3 Tabu Rules for the Array-Swap-Index Neighborhood. The array-swap-index neighborhood, as the name suggests, is a neighborhood over arrays that is generated by swapping pairs of elements based on their positions. An array is a fixed-size collection of elements of some sort E that is indexed by a contiguous subset of the natural numbers, beginning at zero (say); a specification for such arrays can be found in (80). Elements in the array are accessed by their position, or index. The index has to be within the bounds of the array, so in a Slang spec a subsort would be used to enforce this condition. For this discussion, the array element of A at index i will be denoted $A(i)$ and bounds checking will be left implicit. The invariant maintained by this neighborhood is the contents of the array irrespective of position. That is, if the array were converted to a bag, all neighbors of a given array would generate the same bag. The neighborhood is perfect over the set of all arrays whose contents are a particular bag, and is symmetric. The transform variable is a quotient sort. The base sort is a pair of natural numbers $\langle i, j \rangle$. The equivalence is that $\langle i, j \rangle \cong \langle j, i \rangle$. The quotient makes the neighborhood unique; one could accomplish the same thing using a subsort defined by $i \leq j$. A transform variable $\langle i, j \rangle$ is legal if $i \neq j$ and $A(i) \neq A(j)$; if either of these conditions were not true, the swap would not change the solution.

Lock-Out Rule for Array-Swap-Index Neighborhood

```

sort  $T$ 
sort-axiom  $T = \text{Nat}, E, \text{Nat}, E$ 

op  $\text{GetTabu} : \text{Bag}, \text{Array}, C, \text{Array} \rightarrow T$ 
definition of  $\text{GetTabu}$  is
  axiom  $\forall (B : \text{Bag}, A : \text{Array}, i : \text{Nat}, j : \text{Nat}, \text{new\_A} : \text{Array})$ 
     $(\text{GetTabu}(B, A, \langle i, j \rangle, \text{new\_A}) = \langle i, A(i), j, A(j) \rangle)$ 
end-definition

op  $\text{HasTabu} : \text{Bag}, \text{Array}, C, \text{Array}, T \rightarrow \text{Boolean}$ 
definition of  $\text{HasTabu}$  is
  axiom  $\forall (B : \text{Bag}, A : \text{Array}, c : C, A' : \text{Array}, i : \text{Nat}, d : E, j : \text{Nat}, e : E)$ 
     $(\text{HasTabu}(B, A, c, A', \langle i, d, j, e \rangle) \Leftrightarrow A'(i) = d \wedge A'(j) = e)$ 
end-definition

```

Lock-In Rule for Array-Swap-Index Neighborhood

```

sort  $T$ 
sort-axiom  $T = \text{Nat}, E, \text{Nat}, E$ 

op  $\text{GetTabu} : \text{Bag}, \text{Array}, C, \text{Array} \rightarrow T$ 
definition of  $\text{GetTabu}$  is
  axiom  $\forall (B : \text{Bag}, A : \text{Array}, i : \text{Nat}, j : \text{Nat}, \text{new\_A} : \text{Array})$ 
     $(\text{GetTabu}(B, A, \langle i, j \rangle, \text{new\_A}) = \langle i, \text{new\_A}(i), j, \text{new\_A}(j) \rangle)$ 
end-definition

op  $\text{HasTabu} : \text{Bag}, \text{Array}, C, \text{Array}, T \rightarrow \text{Boolean}$ 
definition of  $\text{HasTabu}$  is
  axiom  $\forall (B : \text{Bag}, A : \text{Array}, c : C, A' : \text{Array}, i : \text{Nat}, d : E, j : \text{Nat}, e : E)$ 
     $(\text{HasTabu}(B, A, c, A', \langle i, d, j, e \rangle) \Leftrightarrow \neg(A'(i) = d) \vee \neg(A'(j) = e))$ 
end-definition

```

Figure 112. Lock-Out and Lock-In Rules for Array-Swap-Index Neighborhood

The lock-out and lock-in strategies focus on how two neighboring solutions differ. The move $\langle B, A, \langle i, j \rangle, \text{new\_A} \rangle$, where B is a bag of elements, implies that A and new_A are the same except for

$$A(i) \neq \text{new\_A}(i) \wedge A(j) \neq \text{new\_A}(j)$$

The lock-out rule forbids future solutions A' from resembling A with respect to these changes. The lock-in rule requires future solutions to resemble new_A . These two rules are defined as shown in Figure 112.

Figure 113 shows rules generated from the lock-out and lock-in rules by separating the tests in *HasTabu* and recombining them in various ways. On the left, the original lock-out rule is at the top and is successively relaxed (made more restrictive) toward the bottom. On the right, the original lock-in rule is at the bottom and is successively strengthened (made less restrictive) toward the top. Note that the elements d and e in each diagram refer to different elements: $A(i)$ vs. $new\_A(i)$ and $A(j)$ vs. $new\_A(j)$, respectively. Taking this into account, each lock-out rule is stronger (less restrictive) than the corresponding lock-in rule. For example, the lock-out rule $A(i) = d$ forbids $A(i)$ from assuming a particular value, while the lock-in rule $\neg(A(i) = d)$ forbids all values *except* one, including the one forbidden by the lock-out rule. By transitivity, this makes the original lock-out rule the least restrictive of them all and the original lock-in rule the most restrictive; all are sufficient to prevent solutions from recurring. The remaining rule pairs are not comparable. For example, the bottom lock-out rule forbids some moves allowed by the top lock-in rule and allows some it permits. The lock-out rule will permit future solutions to differ from new_A at both i and j positions as long as neither changes back to its value in A , but the lock-in rule forbids all such future solutions. On the other hand, the lock-in rule allows future solutions to assume their old values at i or at j , but not both, whereas the lock-out rule forbids such solutions. Similar arguments can be made for the incomparability of other pairs of lock-in and lock-out rules that are not compared in Figure 113 either explicitly or by transitivity.

Each transform variable is its own inverse, regardless of the solution to which it is applied. The inverse move rule is again not strong enough to prevent solutions from recurring. For example, the series of transforms

$$\langle j, l \rangle \langle i, k \rangle \langle j, k \rangle \langle i, l \rangle \langle k, l \rangle \langle i, j \rangle$$

regenerates a solution without repeating any moves. No shorter series does this, however, so the rule may be a reasonable heuristic. The inverse move rule is strong enough to prevent states from recurring, because the repeat move rule is identical to it.

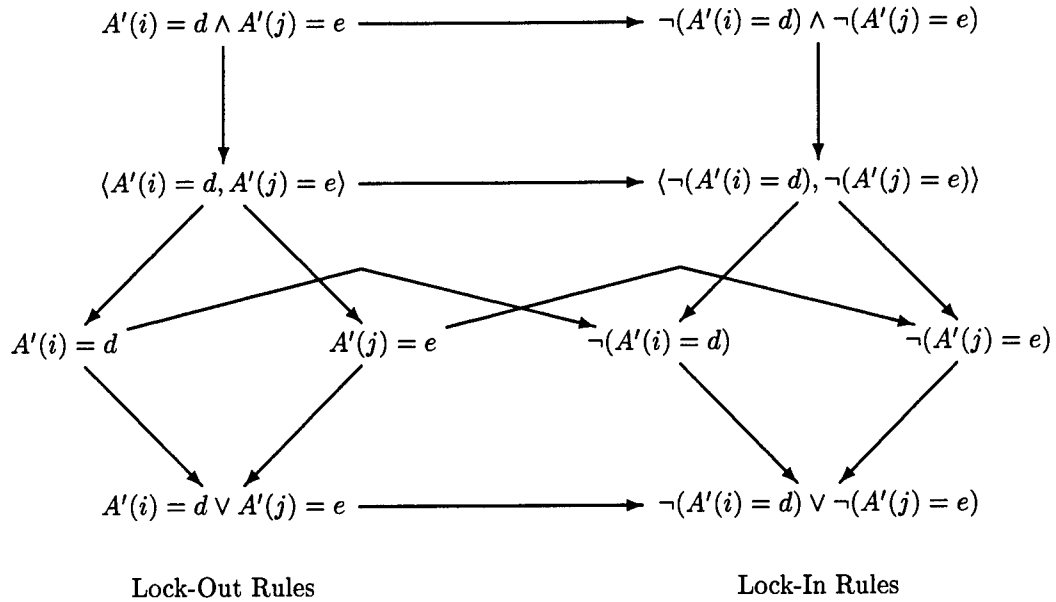


Figure 113. Tabu Rules for Array-Swap-Index Derived from the Lock-Out and Lock-In Rules

The tuple used in the inverse move rule can be split and recombined to generate the diagram of rules shown in Figure 114 (the rules are not quite as shown since C is actually a quotient sort; completeness has been traded for clarity). The inverse move rule is at the top. It is not equivalent to lock-out or lock-in, since each of these prevents solutions from recurring. The remaining rules, however, are equivalent to lock-in rules in the corresponding positions. In particular, the disjunction at the bottom is the lock-in rule itself. This is a consequence of the close relationship between the solution attributes used by lock-in and the transform variables, specifically that the value of an array at position i can only be changed by a transform containing i . The lock-in rule can be optimized to take advantage of this relationship, but it is interesting to note that doing so changes the set of derived rules and even causes it to include one that no longer prevents solution cycles.

For completeness one can try to decompose the 4-tuples used by the lock-out and lock-in rules in ways not yet considered. The elements d and e by themselves do not mean much: all solutions include them somewhere, so without an index in mind they do not suggest any meaningful test.

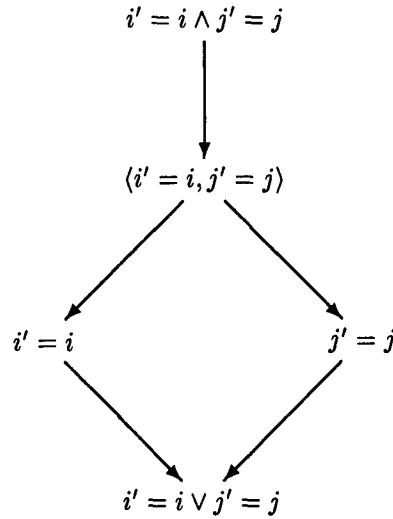


Figure 114. Tabu Rules for Array-Swap-Index Derived from the Inverse Move Rule

Considering i and j as move attributes leads to the hierarchy of rules generated by inverse move, or the optimized lock-in rule. This exhausts the possibilities without producing any new rules.

7.1.2.4 Tabu Rules for the Map-Set-Var-to-Val Neighborhood. Map-set-var-to-val is a perfect, symmetric neighborhood over finite maps whose domain is a specified set Q and whose range is a subset of a specified set R . A transform variable is a pair, $\langle a, b \rangle$, where $a \in Q$ and $b \in R$. A transform is legal for a solution M if $M(a) \neq b$. The result of the transform is a new map M' such that M' is the same as M everywhere except at a , where $M'(a) = b$.

The lock-out rule for map-set-var-to-val forbids future solutions from resembling M , specifically at a . The lock-in rule requires future solutions to resemble M' , specifically at a . These rules are shown in Figure 115. The lock-out rule, which forbids only one value at a , is again stronger and hence less restrictive than the lock-in rule, which forbids all values except one. The *HasTabu* tests are equalities over scalar data, so no decompositions suggest themselves. The sort used as T is a tuple. Treating the components as separate attributes yields two possibilities for each rule. For lock-out, one can forbid all moves that assign any value at a , which is equivalent to lock-in and may be used to optimize it, or one can forbid all moves that assign the value b to any domain

element, which is likely to be too restrictive, as is the disjunctive combination of both rules. For lock-in, one can require all future moves to assign a value at a , which is much too restrictive and yet does not prevent cycling of any kind, or one can require all future moves to assign the value b to some location, which prevents solution cycling but is equally absurd.

The inverse move of $\langle a, b \rangle$ applied to M is made by applying $\langle a, M(a) \rangle$. This is the first example we have seen where the inverse move depends on the solution as well as the transform variable. In this case, the corresponding inverse move tabu rule is equivalent to lock-out. The repeat move rule is distinct and generates its own set of derivative rules, but again they are not very interesting.

7.1.2.5 Tabu Rules for the Array-Insert-Element Neighborhood. Array-insert-element is another perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements. A transform variable is a pair of indices, $\langle i, j \rangle$, in the bounds of the array and distinct. A transform variable is applied to an array by moving the element at index j to index i and moving the elements at positions i through $j - 1$ up (if $i < j$) or elements at $j + 1$ through i up (if $j < i$) one index position. Thus the effects of a move are not as well localized as they were for the neighborhoods above: an entire range of index positions is affected.

The inverse of applying $\langle i, j \rangle$ is to apply $\langle j, i \rangle$, regardless of the arrays involved. This does not prevent cycling. Indeed, once intervening moves have shuffled and rearranged the array between indices i and j , the effect of applying the transform $\langle j, i \rangle$ is likely to bear little resemblance to the move it once inverted.

Other tabu rules that focus on absolute position in the array are likely to be similarly ineffective, because absolute position is so easily and globally affected by moves. Lock-in rules that require an element to remain at a particular index, for example, are likely to be too restrictive because they make many moves tabu. Rules that require an element to avoid certain indices seem, like inverse move, to forbid the wrong set of moves, even though all the standard guarantees hold.

Lock-Out Rule for Map-Set-Var-to-Val Neighborhood

```

sort  $T$ 
sort-axiom  $T = C$ 

op  $GetTabu : Move \rightarrow T$ 
definition of  $GetTabu$  is
  axiom  $\forall (n : Move, x : (Set, Set), M : Map, a : Dom, b : Cod, new\_M : Map,$ 
     $M-at-a : (Map, Dom) \mid defined-at?)$ 
     $(\langle x, M, \langle a, b \rangle, new\_M \rangle = (\mathbf{relax} \text{ LegalMove})(n)$ 
       $\wedge \langle M, a \rangle = (\mathbf{relax} \text{ defined-at?})(M-at-a)$ 
       $\Rightarrow (GetTabu(n) = \langle a, map-apply(M-at-a) \rangle))$ 
end-definition

op  $HasTabu : Move, T \rightarrow Boolean$ 
definition of  $HasTabu$  is
  axiom  $\forall (n : Move, x : (Set, Set), M : Map, c : C, M' : Map, a : Dom, b : Cod,$ 
     $M'-at-a : (Map, Dom) \mid defined-at?)$ 
     $(\langle x, M, c, M' \rangle = (\mathbf{relax} \text{ LegalMove})(n)$ 
       $\wedge \langle M', a \rangle = (\mathbf{relax} \text{ defined-at?})(M'-at-a)$ 
       $\Rightarrow (HasTabu(n, \langle a, b \rangle) \Leftrightarrow map-apply(M'-at-a) = b))$ 
end-definition

```

Lock-In Rule for Map-Set-Var-to-Val Neighborhood

```

sort  $T$ 
sort-axiom  $T = C$ 

op  $GetTabu : Move \rightarrow T$ 
definition of  $GetTabu$  is
  axiom  $\forall (n : Move, x : (Set, Set), M : Map, c : C, new\_M : Map)$ 
     $(\langle x, M, c, new\_M \rangle = (\mathbf{relax} \text{ LegalMove})(n)$ 
       $\Rightarrow GetTabu(n) = c)$ 
end-definition

op  $HasTabu : Move, T \rightarrow Boolean$ 
definition of  $HasTabu$  is
  axiom  $\forall (n : Move, x : (Set, Set), M : Map, c : C, M' : Map, a : Dom, b : Cod,$ 
     $M'-at-a : (Map, Dom) \mid defined-at?)$ 
     $(\langle x, M, c, M' \rangle = (\mathbf{relax} \text{ LegalMove})(n)$ 
       $\wedge \langle M', a \rangle = (\mathbf{relax} \text{ defined-at?})(M'-at-a)$ 
       $\Rightarrow (HasTabu(n, \langle a, b \rangle) \Leftrightarrow \neg(map-apply(M'-at-a) = b)))$ 
end-definition

```

Figure 115. Lock-Out and Lock-In Rules for Map-Set-Var-to-Val Neighborhood

If the element at j is moved to index i , for example, and a set of intervening moves over other indices happens to shuffle the element back to the vicinity of j , it seems unlikely that avoiding the moves that return it to its original position will guide the search effectively.

The relevant attributes for tabu rules are more likely to be those based on viewing the array as a kind of sequence rather than as a kind of map. For example, one result of a move is that $A(i)$ and $A(j)$ are adjacent in the new array, where in the old one $A(i)$ and $A(j)$ each had some other neighbor. Other than at i and j , adjacency did not change. Future moves may change the relative positions of $A(i)$ and $A(j)$ but will not separate them unless i or an adjacent index is in the transform variable. A possible lock-out rule is to forbid solutions that result in $A(j)$ being adjacent to, say, $A(j + 1)$, its old neighbor. A lock-in rule might require $A(i)$ and $A(j)$ to remain adjacent, now that they are.

Which kind of attribute, position or adjacency, makes more sense depends strongly on the problem to which this neighborhood is applied, in particular how the cost function is affected by changes in position vs. adjacency. What is noteworthy about this neighborhood, then, is not the variety of tabu rules one can choose from, but rather that tabu rules are hard to conceive and their behavior hard to predict. One might speculate that this neighborhood is inherently not well suited to the technique, or at least is more sensitive to problem structure than the ones considered previously.

7.1.3 Aspiration Criteria. Aspiration criteria provide a means to free up a search by overriding tabu rules. The intent is to identify conditions under which a move is known not to cause a solution or state to be revisited even though a tabu rule indicates that it might. Like tabu rules, the guarantees provided by aspiration criteria vary.

Most aspiration criteria can be described in a theoretical framework very similar to that developed for tabu rules. Aspiration criteria identify move attributes and associate *aspiration levels* with moves based on these attributes. A move satisfies its aspiration criteria if its aspiration

level exceeds the level for other moves with the same attribute value. Aspiration levels are most often costs, so for example a move might satisfy its aspiration level if the cost of the solution it moves to is better than the cost associated with other moves with its attribute.

Aspiration by default is atypical. This rule says that if all moves are tabu, choose the move that is least tabu, in the sense that it is classified tabu by elements that have been on the tabu list the longest and hence would expire the soonest (27). There are no levels or move attributes involved, so this rule does not fit the standard framework. It is best implemented directly in the program scheme, where the tabu list data structure is introduced. Aspiration by default is universally applicable to tabu search and is widely used.

To support the general principles of aspiration, the extensions shown in Figure 116 are made to *TabuSearch*. *A* is the sort of the attribute used to group moves into classes. The operation *GetAspiration* extracts an attribute associated with a move, and *HasAspiration* checks whether a move satisfies or possesses an attribute. *ALevel* is the sort of aspiration levels. The operation *GetLevel* associates a level with a move and *HasLevel* determines whether a move exceeds its level. *ALevel* may have a total order, which it does if costs are used, in which case the level associated with a move by *GetLevel* is typically interpreted as the interval of all values less than that level and *HasLevel* is a simple less-than test. In other cases *ALevel* may be a set of values from some base sort and *HasLevel* a test for membership. The framework adopted here is flexible enough to support these and other conceptions of a level.

The property that aspiration criteria are intended to satisfy is described by the axiom in *TASearch*. The form chosen is the contrapositive of the form used in the corresponding condition for tabu rules, where move equality implied the tabu condition. This is simply a reflection of the preferred interpretation of the operations: *HasTabu* is true when a move is forbidden, whereas *HasLevel* is true when a move is allowed. It describes a limited form of state cycle prevention for aspiration. It says the aspiration criteria imply that the current move is not the same as a previous

```

spec TASearch is
  import TabuSearch

  sort A
  op GetAspiration : Move  $\rightarrow$  A
  op HasAspiration : Move, A  $\rightarrow$  Boolean

  sort ALevel
  op GetLevel : Move  $\rightarrow$  ALevel
  op HasLevel : Move, ALevel  $\rightarrow$  Boolean

  axiom  $\forall (x : D, z : R, c : C, z' : R, z'' : R, c'' : C, z''' : R, m : \text{Move}, n : \text{Move})$ 
    ( $\langle x, z, c, z' \rangle = (\text{relax LegalMove})(m) \wedge N^*(x, z', z'')$ 
       $\wedge \langle x, z'', c'', z''' \rangle = (\text{relax LegalMove})(n)$ 
       $\wedge \text{HasAspiration}(n, \text{GetAspiration}(m))$ 
       $\wedge \text{HasLevel}(n, \text{GetLevel}(m))$ 
       $\Rightarrow \neg(m = n))$ 

end-spec

```

Figure 116. Basic Elements of Aspiration Criteria

move with the same attribute. This condition is not as strong as the conditions for tabu rules. If a move is not tabu by any previous move, then the cycle prevention axiom guarantees that the move has not been made before and the strong cycle prevention axiom guarantees that the to-solution has never been visited before. The condition for aspiration criteria only compares aspiration levels of moves that share an attribute with a particular move, so the only guarantee is that the move is not any of the moves in this class. Since not all moves are compared, no stronger guarantee is possible.

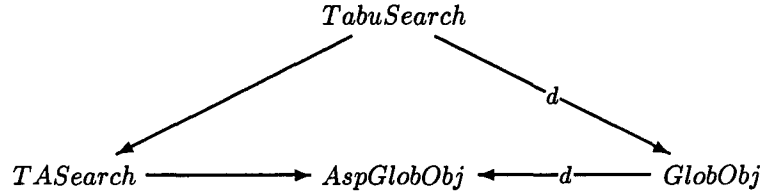
Criteria that use costs as levels, at least, can be described as *forward-looking* or *backward-looking*. A forward-looking criterion compares the cost of a solution moved to by a previous move to the cost of the solution moved to by the current trial move. The intent is to prevent moving to a solution *to which* we have moved before. A backward-looking criterion compares the cost of a solution moved from by a previous move to the cost of the solution moved to by the current trial move. The intent is to prevent moving back to a solution *from which* we have moved before. In each case we are trying to avoid revisiting a solution, but the means by which we do this differ slightly in these two cases. In general, any from-attribute used as a level specifies a backward-looking criterion

and any to-attribute specifies a forward-looking one. Using levels other than costs, such as hash functions, is suggested in the literature but not illustrated (e.g., (27)).

A very commonly used aspiration criterion, called *aspiration by global objective* in (27), checks whether a trial move is to a solution better than all solutions previously visited. There is no attribute used by this criterion that distinguishes one move from another, or rather one can think of it as using a universal attribute that all moves share simply by being moves. Aspiration by global objective is formalized in Figure 117. The attribute sort A is the empty product sort. This sort has a single element, denoted by the empty tuple, $\langle \rangle$. All moves possess this attribute. Levels are costs. The level of a move is the cost of the solution it moves to. A move satisfies a specified level by moving to a solution of lower cost. If a move satisfies the level of all moves that share its attribute, then it is distinct from all of them. Since the attribute used is shared by all moves, the move is to a solution that has never before been visited. This is a much stronger guarantee than that described by the condition above. The definitions added to *TabuSearch* could have been done in one spec rather than two, but we chose to separate them to emphasize that this aspiration criterion in effect does not use move attributes but does use levels.

Aspiration by global objective as shown in Figure 117 is forward-looking. The corresponding backward-looking criterion would be the same except *GetLevel* would extract $Cost(x, z)$. The effect would be nearly the same, too, except the cost of the current solution would not be considered when evaluating a candidate move because no move has left it yet (unless the search has cycled). The forward-looking criterion fails to consider the cost of the initial solution, since no move was made to it, but this is easily fixed by a suitable initialization in the program scheme. Criteria based on attributes that are not universal show a greater difference between forward-looking and backward-looking variants.

Figure 118 shows a forward-looking aspiration criterion that uses solution cost as the attribute and as the aspiration level. This criterion can be paraphrased as



```

spec GlobObj is
  import TabuSearch

  sort ALevel
  sort-axiom ALevel =  $\mathcal{R}$ 

  op GetLevel : Move  $\rightarrow$  ALevel
  definition of GetLevel is
    axiom  $\forall (x : D, z : R, c : C, z' : R, m : \text{Move})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax LegalMove})(m))$ 
       $\Rightarrow \text{GetLevel}(m) = \text{Cost}(x, z')$ 
  end-definition

  op HasLevel : Move, ALevel  $\rightarrow$  Boolean
  definition of HasLevel is
    axiom  $\forall (x : D, z : R, c : C, z' : R, m : \text{Move}, l : \text{ALevel})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax LegalMove})(m))$ 
       $\Rightarrow (\text{HasLevel}(m, l) \Leftrightarrow \text{Cost}(x, z') < l)$ 
  end-definition
end-spec

spec AspGlobObj is
  import GlobObj

  sort A
  sort-axiom A =  $()$ 

  op GetAspiration : Move  $\rightarrow$  A
  definition of GetAspiration is
    axiom  $\forall (m : \text{Move}) (\text{GetAspiration}(m) = ())$ 
  end-definition

  op HasAspiration : Move, A  $\rightarrow$  Boolean
  definition of HasAspiration is
    axiom  $\forall (m : \text{Move}) \text{HasAspiration}(m, ())$ 
  end-definition
end-spec

interpretation AspByGlobObj : TASearch  $\Rightarrow$  GlobObj is
  mediator AspGlobObj
  domain-to-mediator {}
  codomain-to-mediator import-morphism

```

Figure 117. Aspiration by Global Objective

In the past I made a move from a solution of cost $Cost(x, z)$ to one of cost $Cost(x, z')$. Now I am again at a solution of cost $Cost(x, z)$. If I move to a solution of cost better than $Cost(x, z')$, then I know this solution is not the same as the one before.

This criterion divides moves into equivalence classes based on the costs of the solutions they move to. It helps to prevent repeating a move, and is often used in conjunction with the repeat move tabu rule, but it does not prevent visiting a solution by more than one trajectory if the trajectories are via solutions of differing costs.

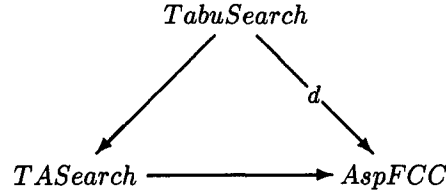
Figure 119 shows backward-looking aspiration with cost as the attribute and the aspiration level. It can be paraphrased as

In the past I made a move from a solution of cost $Cost(x, z)$ to one of cost $Cost(x, z')$. I am now at a solution of cost $Cost(x, z)$. If I move to a solution of cost better than $Cost(x, z)$, then I know I have not reversed the previous move.

This again partitions moves by cost, but this time based on from-solutions. It is usually used in conjunction with the inverse move strategy and may also be suitable for other strategies that prevent solutions from recurring, such as lock-out and lock-in. Again, this aspiration criterion does not prevent a move from being approached from multiple trajectories; (25) illustrates the problem with an example and suggests using a heuristic to mitigate its effects.

Just as tabu search often combines the repeat move and inverse move strategies, so too are the above two aspiration criteria used in combination (20, 25, 38). That is, the solution moved to by a candidate move that is tabu must have better cost than all moves made to solutions of cost equal to the current solution and all moves made from solutions of cost equal to the current solution to be admissible, trying to prevent both repeat and inverse moves simultaneously. In effect, the two criteria are being applied conjunctively: both must be satisfied to override a tabu status imposed by either of the two tabu rules. Alternatives not considered in the literature are to combine them disjunctively, or to apply only the one that corresponds to the particular tabu rule violated.

The efficacy of these two aspiration criteria seems doubtful to this author, who admittedly has no practical experience in applying them. Partitioning moves based on cost seems to bear



```

spec AspFCC is
  import TabuSearch

  sort A
  sort-axiom A = R

  op GetAspiration : Move → A
  definition of GetAspiration is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m))$ 
       $\Rightarrow GetAspiration(m) = Cost(x, z)$ 
  end-definition

  op HasAspiration : Move, A → Boolean
  definition of HasAspiration is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move, a : A)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m))$ 
       $\Rightarrow (HasAspiration(m, a) \Leftrightarrow a = Cost(x, z))$ 
  end-definition

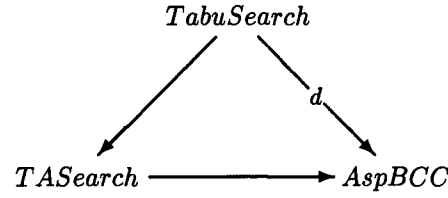
  sort ALevel
  sort-axiom ALevel = R

  op GetLevel : Move → ALevel
  definition of GetLevel is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m))$ 
       $\Rightarrow GetLevel(m) = Cost(x, z')$ 
  end-definition

  op HasLevel : Move, ALevel → Boolean
  definition of HasLevel is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : Move, l : ALevel)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } LegalMove)(m))$ 
       $\Rightarrow (HasLevel(m, l) \Leftrightarrow Cost(x, z') < l)$ 
  end-definition
end-spec

```

Figure 118. Forward-Looking, Cost-Level Aspiration With Cost as Attribute



```

spec AspBCC is
  import TabuSearch

  sort A
  sort-axiom  $A = \mathcal{R}$ 

  op GetAspiration : Move  $\rightarrow$  A
  definition of GetAspiration is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : \text{Move})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } \text{LegalMove})(m))$ 
       $\Rightarrow \text{GetAspiration}(m) = \text{Cost}(x, z')$ 
  end-definition

  op HasAspiration : Move, A  $\rightarrow$  Boolean
  definition of HasAspiration is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : \text{Move}, a : A)$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } \text{LegalMove})(m))$ 
       $\Rightarrow (\text{HasAspiration}(m, a) \Leftrightarrow a = \text{Cost}(x, z))$ 
  end-definition

  sort ALevel
  sort-axiom  $A\text{Level} = \mathcal{R}$ 

  op GetLevel : Move  $\rightarrow$  ALevel
  definition of GetLevel is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : \text{Move})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } \text{LegalMove})(m))$ 
       $\Rightarrow \text{GetLevel}(m) = \text{Cost}(x, z)$ 
  end-definition

  op HasLevel : Move, ALevel  $\rightarrow$  Boolean
  definition of HasLevel is
    axiom  $\forall(x : D, z : R, c : C, z' : R, m : \text{Move}, l : A\text{Level})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax } \text{LegalMove})(m))$ 
       $\Rightarrow (\text{HasLevel}(m, l) \Leftrightarrow \text{Cost}(x, z') < l)$ 
  end-definition
end-spec

```

Figure 119. Backward-Looking, Cost-Level Aspiration With Cost as Attribute

little relationship to the structure of most problems. If a solution can be generated by more than one prior solution, it seems likely that these solutions will have different costs. Moves involving solutions that have the same cost, on the other hand, seem likely to be unrelated to each other. Significantly, perhaps, more recent papers that describe aspiration criteria in general and survey various approaches do not discuss these criteria (27).

Figure 120 is a forward-looking criterion that uses a tabu rule to identify the attribute and costs as levels. It can be paraphrased as

Previously I made a move to a solution of cost $Cost(x, z')$. I am now considering a move classified tabu by that move. If it is to a solution of better cost, then it is not to the same solution z' as before.

Here the aspiration criterion clearly relates to the tabu status: among the moves that would classify the trial move as tabu, the trial move is shown to be distinct from them. This criterion should work well with repeat move, which is itself a forward-looking rule. It probably would not work well with inverse move, lock-out or lock-in, which are better served by the next one.

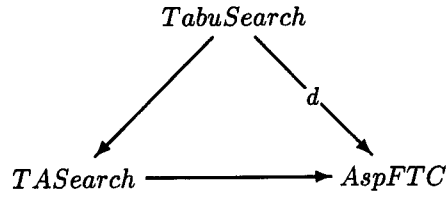
Figure 121 shows a backward-looking criterion that uses a tabu rule to define the attribute and costs as levels. It can be paraphrased as

Previously I made a move from a solution of cost $Cost(x, z)$. I am now considering a move classified tabu by that move. If it is to a solution of better cost, then I am not moving back to the solution z .

This works well with backward-looking tabu rules: those that try to prevent solutions from recurring. If the tabu rule guarantees strong cycle prevention, using this criterion will not ruin it.

For all of these aspiration criteria that use costs as levels, if a solution is found that is a new best solution, it would satisfy any of them. Thus it is not necessary in a program scheme to combine aspiration by global objective with any of the others because it would just introduce redundant checking.

Figure 122 shows *aspiration by search direction* (27). This rule uses a tabu attribute as the aspiration attribute and defines the level as a boolean value indicating whether the move was



```

spec AspFTC is
  import TabuSearch

  sort A
  sort-axiom A = T

  op GetAspiration : Move → A
  definition of GetAspiration is
    axiom GetAspiration = GetTabu
  end-definition

  op HasAspiration : Move, A → Boolean
  definition of HasAspiration is
    axiom HasAspiration = HasTabu
  end-definition

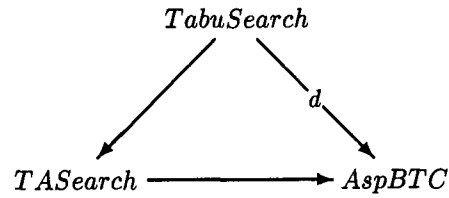
  sort ALevel
  sort-axiom ALevel =  $\mathcal{R}$ 

  op GetLevel : Move → ALevel
  definition of GetLevel is
    axiom  $\forall (x : D, z : R, c : C, z' : R, m : \text{Move})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax LegalMove})(m))$ 
       $\Rightarrow \text{GetLevel}(m) = \text{Cost}(x, z')$ 
  end-definition

  op HasLevel : Move, ALevel → Boolean
  definition of HasLevel is
    axiom  $\forall (x : D, z : R, c : C, z' : R, m : \text{Move}, l : \text{ALevel})$ 
       $(\langle x, z, c, z' \rangle = (\text{relax LegalMove})(m))$ 
       $\Rightarrow (\text{HasLevel}(m, l) \Leftrightarrow \text{Cost}(x, z') < l)$ 
  end-definition
end-spec

```

Figure 120. Forward-Looking, Cost-Level Aspiration With Tabu Attribute



```

spec AspBTC is
  import TabuSearch

  sort A
  sort-axiom A = T

  op GetAspiration : Move → A
  definition of GetAspiration is
    axiom GetAspiration = GetTabu
  end-definition

  op HasAspiration : Move, A → Boolean
  definition of HasAspiration is
    axiom HasAspiration = HasTabu
  end-definition

  sort ALevel
  sort-axiom ALevel = R

  op GetLevel : Move → ALevel
  definition of GetLevel is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move)
      ((x, z, c, z') = (relax LegalMove)(m)
       ⇒ GetLevel(m) = Cost(x, z))
  end-definition

  op HasLevel : Move, ALevel → Boolean
  definition of HasLevel is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move, l : ALevel)
      ((x, z, c, z') = (relax LegalMove)(m)
       ⇒ (HasLevel(m, l) ⇔ Cost(x, z') < l))
  end-definition
end-spec

```

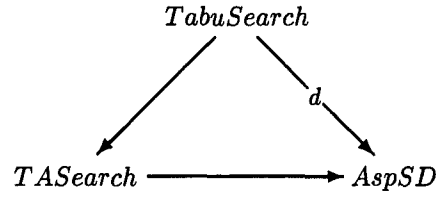
Figure 121. Backward-Looking, Cost-Level Aspiration with Tabu Attribute

improving or not. The current trial move satisfies its aspiration criterion if the move that classifies it tabu was improving and the trial move is also improving. The rationale is that if the search direction (improving or non-improving) is different for the previous move and the trial move then the trial move could be reversing the previous move, but if both moves are improving then the likelihood of the trial move being to a new solution is much greater.

In multi-list environments, each tabu rule can have its own separate set of aspiration criteria, or criteria that are independent of tabu attributes can be shared. Aspiration criteria are tested before list results are combined.

7.1.4 Program Scheme for Tabu Search. Figures 123 through 125 present a domain theory for single-list tabu search with aspiration criteria. A tabu list will be represented by a sequence, so Figure 123 extends *BasicSeq* from Chapter II with some standard sequence operations. In Figure 124, *TAS-Import* extends *TASearch* with several new data types: sequences of tabu elements, maps from aspiration attributes to aspiration levels, and sets of moves. It also incorporates the feasibility axiom for neighborhoods, which this program scheme requires to guarantee a feasible solution is returned. In Figure 125, *TASProgBase* extends *TAS-Import* with tunable parameters and operations representing design decisions that need to be made once this program scheme has been selected.

The *InitSol* operation finds an initial feasible solution based on the input and initial solution parameters of sort *ISP*; *InitISP* finds an initial parameter. The predicate *LegalISP* has been dropped for simplicity: the designer is free to define *ISP* as a subsort if desired, or by any other means. *MaxTries* computes the number of searches to perform. *MaxNoChamp* is the maximum number of search steps that will be executed without finding a new best solution or *champion*. This resource limitation is incorporated into the stopping criteria. *GOTest* is again a test for global optimality and is used as another stopping criterion.



```

spec AspSD is
  import TabuSearch

  sort A
  sort-axiom A = T

  op GetAspiration : Move → A
  definition of GetAspiration is
    axiom GetAspiration = GetTabu
  end-definition

  op HasAspiration : Move, A → Boolean
  definition of HasAspiration is
    axiom HasAspiration = HasTabu
  end-definition

  sort ALevel
  sort-axiom ALevel = Boolean

  op GetLevel : Move → ALevel
  definition of GetLevel is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move)
      ((x, z, c, z') = (relax LegalMove)(m)
       ⇒ GetLevel(m) = (Cost(x, z') < Cost(x, z)))
  end-definition

  op HasLevel : Move, ALevel → Boolean
  definition of HasLevel is
    axiom ∀(x : D, z : R, c : C, z' : R, m : Move, l : ALevel)
      ((x, z, c, z') = (relax LegalMove)(m)
       ⇒ (HasLevel(m, l) ⇔ Cost(x, z') < Cost(x, z) ∧ l))
  end-definition
end-spec

```

Figure 122. Aspiration By Search Direction

```

spec Seq is
  import BasicSeq

  op singleton : E → Seq
  definition of singleton is
    axiom  $\forall (e : E) (singleton(e) = prepend(e, empty-seq))$ 
  end-definition

  op append : E, Seq → Seq
  definition of append is
    axiom  $\forall (e : E) (append(e, empty-seq) = prepend(e, empty-seq))$ 
    axiom  $\forall (d : E, e : E, S : Seq) (append(e, prepend(d, S)) = prepend(d, append(e, S)))$ 
  end-definition

  op concat : Seq, Seq → Seq
  definition of concat is
    axiom  $\forall (S : Seq) (concat(S, empty-seq) = S)$ 
    axiom  $\forall (e : E, S : Seq, T : Seq) (concat(S, prepend(e, T)) = concat(append(e, S), T))$ 
  end-definition

  op delete : E, Seq → Seq
  definition of delete is
    axiom  $\forall (e : E) (delete(e, empty-seq) = empty-seq)$ 
    axiom  $\forall (e : E, S : Seq) (delete(e, prepend(e, S)) = delete(e, S))$ 
    axiom  $\forall (d : E, e : E, S : Seq)$ 
       $(\neg(d = e) \Rightarrow delete(e, prepend(d, S)) = prepend(d, delete(e, S)))$ 
  end-definition

  op in-seq : E, Seq → Boolean
  definition of in-seq is
    axiom  $\forall (e : E) \neg in-seq(e, empty-seq)$ 
    axiom  $\forall (d : E, e : E, S : Seq) (in-seq(e, prepend(d, S)) \Leftrightarrow e = d \vee in-seq(e, S))$ 
  end-definition
end-spec

spec ExtSeq is
  import Seq, Nat

  op length : Seq → Nat
  definition of length is
    axiom  $size(empty-seq) = zero$ 
    axiom  $\forall (e : E, S : Seq) (length(prepend(e, S)) = nat-of-pos(succ(length(S))))$ 
  end-definition

  op truncate : Seq, Nat → Seq
  definition of truncate is
    axiom  $\forall (S : Seq) (truncate(S, zero) = empty-seq)$ 
    axiom  $\forall (k : Nat) (truncate(empty-seq, k) = empty-seq)$ 
    axiom  $\forall (e : E, S : Seq, k : Nat)$ 
       $(truncate(prepend(e, S), nat-of-pos(succ(k))) = prepend(e, truncate(S, k)))$ 
  end-definition
end-spec

```

Figure 123. Spec for Sequences


```

spec TASProgBase is
  import TAS-Import

  sort ISP

  op InitISP : D → ISP

  op InitSol : D, ISP → R, ISP
  axiom  $\forall(x : D, p : ISP) (I(x) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$ 

  op MaxTries : D → Pos

  op MaxNoChamp : D → Nat

  op TabuLen : D → Nat

  op Optimal : D, R → Boolean
  definition of Optimal is
    axiom  $\forall(x : D, z : R) (Optimal(x, z) \Leftrightarrow$ 
       $\forall(z' : R) (O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z')))$ 
  end-definition

  op GOTest : D, R → Boolean
  axiom  $\forall(x : D, z : R) (GOTest(x, z) \Rightarrow Optimal(x, z))$ 

  op InitAspirationMap : D, R → AspirationMap

  op NewAspirationMap : Move, AspirationMap → AspirationMap
  axiom  $\forall(m : Move, AM : AspirationMap, a : A,$ 
     $AM\text{-}at\text{-}a : (AspirationMap, A) \mid \text{defined-at?},$ 
     $new\_AM\text{-}at\text{-}a : (AspirationMap, A) \mid \text{defined-at?})$ 
     $((AM, a) = (\text{relax defined-at?})(AM\text{-}at\text{-}a)$ 
     $\wedge (NewAspirationMap(m, AM), a) = (\text{relax defined-at?})(new\_AM\text{-}at\text{-}a)$ 
     $\wedge \neg(GetAspiration(m) = a)$ 
     $\Rightarrow \text{map-apply}(AM\text{-}at\text{-}A) = \text{map-apply}(new\_AM\text{-}at\text{-}a))$ 

  op GetCandidateMoves : D, R → MSet
  axiom  $\forall(x : D, z : R, m : Move)$ 
     $(in(m, GetCandidateMoves(x, z))$ 
     $\Rightarrow x = (\text{project } 1)(m) \wedge z = (\text{project } 2)(m))$ 

  op ChooseMove : NE-MSet → Move
  axiom  $\forall(MS : NE\text{-}MSet) in(ChooseMove(MS), MS)$ 
end-spec

```

Figure 125. Domain Theory for Tabu Search with Aspiration, Cont.

TabuLen is the length of the tabu list. By imposing a limit on the size of the list, cycle prevention is not assured even if the tabu strategy used is capable of it. Termination is ultimately controlled by the stopping criteria becoming true even if cycling is occurring. Common approaches to setting the length are to use a constant (7 is a popular choice) or a size parameter based on the input (27). More elaborate program schemes would allow the tabu length to vary during a search.

InitAspirationMap initializes an aspiration map. If aspiration by global objective is used, this would set the sole map entry to the cost of the initial solution. Other aspiration criteria also sometimes provide initial values for the set of possible aspiration attributes. Initialization is not required, however: the empty map is a legitimate initial map for many criteria. *NewAspirationMap* updates the aspiration map with a selected move by adding (or updating) the association between the aspiration attribute of the move and its level. For some aspiration criteria the update is contingent. For example, when costs are used as levels, the level associated with an attribute is changed only if the current move has a better level. Other criteria, such as aspiration by search direction, always store a level for the current move. The axiom in the domain theory says only that *NewAspirationMap* cannot change the map except at the attribute of the specified move.

GetCandidateMoves computes for a given feasible solution a set of moves that will be considered. This can be the entire neighborhood or can be more selective. An operation described below will select from a set of candidate moves the ones that are admissible. *ChooseMove* is a tie-breaker: if two or more moves are considered equally acceptable, this operation selects one.

Figures 126 through 128 show the mediator spec of the program scheme. *InitTabuList* defines the empty sequence as the initial tabu list for all inputs; the signature was chosen to suggest potential dependencies of other methods for selecting an initial list. *NewTabuList* updates the tabu list with a selected move by adding a tabu element to the front of the list and truncating the list to the maximum length. If the list is already shorter than the maximum length, truncation

has no effect. *IsTabu* determines whether a candidate move is tabu, that is, if it possesses any of the attributes on the tabu list.

MeetsAspiration determines whether a move meets its aspiration level, which it does if it meets all the levels prescribed by all the attributes it possesses and for which the map is defined. For most aspiration criteria, including those that use cost as the aspiration attribute, a move will have at most one attribute where the map is defined. For aspiration based on tabu rules, however, a move might be classified tabu by more than one tabu element, in which case it needs to meet all of the corresponding aspiration levels to insure that it is not repeating any previous solution. If, for a particular choice of aspiration criteria and tabu rule, it is known that this cannot happen, *MeetsAspiration* can be optimized to extract the only aspiration attribute a move possesses and simply apply the map to it to get the level. Implicit in the definition of *MeetsAspiration* is that if a move has an attribute that is not assigned a level, it meets its aspiration level. This can reduce the need to initialize the aspiration map before search begins.

GetAdmissibleMoves takes a set of candidate moves, a tabu list and an aspiration map and determines which moves are admissible. Moves are admissible as long as they are not tabu, or if tabu, they meet their aspiration levels. If desired, this operation could be extended to perform aspiration by default. *GetBestMoves* takes a non-empty set of moves and selects those of best cost.

The remainder of the program scheme follows the familiar pattern. *TabuMain* accepts an input and calls *TabuIter* to perform some number of searches. For each, *TabuIter* gets an initial solution, sets up other initial parameters, and calls *TabuStep*; it returns the best final solution found. *TabuStep* searches from neighbor to neighbor until no admissible moves exist, a globally optimal solution is found or the champion fails to change within *MaxNoChamp(x)* steps. At each step the new solution is compared to the champion, replacing it and resetting the count to its maximum value when its cost is better.

```

spec TASProgram is
  import TASProgBase

  op InitTabuList : D, Nat → TabuList
  definition of InitTabuList is
    axiom  $\forall (x : D, len : Nat) (InitTabuList(x, len) = empty-seq)$ 
  end-definition

  op NewTabuList : Move, TabuList, Nat → TabuList
  definition of NewTabuList is
    axiom  $\forall (m : Move, TL : TabuList, len : Nat)$ 
       $(NewTabuList(m, TL, k) = truncate(prepend(GetTabu(m), TL), len))$ 
  end-definition

  op IsTabu : Move, TabuList → Boolean
  definition of IsTabu is
    axiom  $\forall (m : Move) \neg IsTabu(m, empty-seq)$ 
    axiom  $\forall (m : Move, t : T, TL : TabuList)$ 
       $(IsTabu(m, prepend(t, TL)) \Leftrightarrow HasTabu(m, t) \vee IsTabu(m, TL))$ 
  end-definition

  op MeetsAspiration : Move, AspirationMap → Boolean
  definition of MeetsAspiration is
    axiom  $\forall (m : Move) MeetsAspiration(m, empty-map)$ 
    axiom  $\forall (m : Move, AM : AspirationMap, a : A, l : ALevel)$ 
       $(MeetsAspiration(m, map-shadow(AM, a, l))$ 
         $\Leftrightarrow (HasAspiration(m, a) \Rightarrow HasLevel(m, l)) \wedge MeetsAspiration(m, AM))$ 
  end-definition

  op GetAdmissibleMoves : MSet, TabuList, AspirationMap → MSet
  definition of GetAdmissibleMoves is
    axiom  $\forall (TL : TabuList, AM : AspirationMap)$ 
       $(GetAdmissibleMoves(empty-set, TL, AM) = empty-set)$ 
    axiom  $\forall (m : Move, MS : MSet, TL : TabuList, AM : AspirationMap)$ 
       $(\neg IsTabu(m, TL) \vee MeetsAspiration(m, AM)$ 
         $\Rightarrow GetAdmissibleMoves(insert(m, MS), TL, AM)$ 
         $= insert(m, GetAdmissibleMoves(MS, TL, AM)))$ 
    axiom  $\forall (m : Move, MS : MSet, TL : TabuList, AM : AspirationMap)$ 
       $(IsTabu(m, TL) \wedge \neg MeetsAspiration(m, AM)$ 
         $\Rightarrow GetAdmissibleMoves(insert(m, MS), TL, AM)$ 
         $= GetAdmissibleMoves(MS, TL, AM))$ 
  end-definition

```

Figure 126. Mediator Spec for Tabu Search with Aspiration


```

op GetBestMoves : NE-MSet → NE-MSet
definition of GetBestMoves is
  axiom  $\forall(m : \text{Move}, MS : \text{NE-Set}) (in(m, \text{GetBestMoves}(MS)) \Leftrightarrow$ 
     $(in(m, MS) \wedge \forall(x : D, z : R, x' : D, z' : R, n : \text{Move})$ 
       $(x = (\text{project } 1)(m) \wedge z = (\text{project } 4)(m) \wedge in(n, MS)$ 
         $\wedge x' = (\text{project } 1)(n) \wedge z' = (\text{project } 4)(n)$ 
         $\Rightarrow \text{Cost}(x, z) \leq \text{Cost}(x', z'))))$ 
  end-definition

op BetterSol : D, R, R → R
definition of BetterSol is
  axiom  $\forall(x : D, z1 : R, z2 : R) (\text{Cost}(x, z1) \leq \text{Cost}(x, z2) \Rightarrow \text{BetterSol}(x, z1, z2) = z1)$ 
  axiom  $\forall(x : D, z1 : R, z2 : R) (\text{Cost}(x, z2) < \text{Cost}(x, z1) \Rightarrow \text{BetterSol}(x, z1, z2) = z2)$ 
end-definition

op TabuMain : D → R
definition of TabuMain is
  axiom  $\forall(x : D) (\text{TabuMain}(x) = \text{TabuIter}(x, \text{MaxTries}(x), \text{InitISP}(x)))$ 
end-definition

op TabuIter : D, Pos, ISP → R
definition of TabuIter is
  axiom  $\forall(x : D, p : \text{ISP}, z : R)$ 
     $(z = (\text{project } 1)(\text{InitSol}(x, p))$ 
       $\Rightarrow \text{TabuIter}(x, \text{succ}(\text{zero}), p)$ 
       $= \text{TabuStep}(x, z, z, \text{MaxNoChamp}(x), \text{TabuLen}(x),$ 
         $\text{InitTabuList}(x, \text{TabuLen}(x)), \text{InitAspirationMap}(x, z)))$ 
  axiom  $\forall(x : D, k : \text{Pos}, p : \text{ISP}, z : R, p' : \text{ISP}, z' : R, z'' : R)$ 
     $((z, p') = \text{InitSol}(x, p)$ 
       $\wedge z' = \text{TabuStep}(x, z, z, \text{MaxNoChamp}(x), \text{TabuLen}(x),$ 
         $\text{InitTabuList}(x, \text{TabuLen}(x)), \text{InitAspirationMap}(x, z))$ 
       $\wedge z'' = \text{TabuIter}(x, k, p')$ 
       $\Rightarrow \text{TabuIter}(x, \text{succ}(\text{nat-of-pos}(k)), p) = \text{BetterSol}(x, z', z''))$ 
  end-definition

```

Figure 127. Mediator Spec for Tabu Search with Aspiration, Cont.

```

op TabuStep : D, R, R, Nat, Nat, TabuList, AspirationMap → R
definition of TabuStep is
  axiom  $\forall (x : D, curr : R, champ : R, nochamp : Nat, tlen : Nat, TL : TabuList,$ 
     $AM : AspirationMap)$ 
     $(nochamp = zero \vee GOTest(x, curr)$ 
       $\vee GetAdmissibleMoves(GetCandidateMoves(x, curr), TL, AM)$ 
         $= empty-set$ 
       $\Rightarrow TabuStep(x, curr, champ, nochamp, tlen, TL, AM) = champ)$ 
  axiom  $\forall (x : D, curr : R, champ : R, nochamp : Nat, tlen : Nat, TL : TabuList,$ 
     $AM : AspirationMap, MS : NE-MSet, m : Move)$ 
     $(nonzero?(nochamp) \wedge \neg GOTest(x, curr)$ 
       $\wedge GetAdmissibleMoves(GetCandidateMoves(x, curr), TL, AM)$ 
         $= (relax\ nonempty-set?)(MS)$ 
       $\wedge m = ChooseMove(GetBestMoves(MS))$ 
       $\wedge new = (project\ 4)(m)$ 
       $\wedge Cost(x, new) < Cost(x, champ)$ 
       $\Rightarrow TabuStep(x, curr, champ, nochamp, tlen, TL, AM)$ 
         $= TabuStep(x, new, new, MaxNoChamp(x), tlen,$ 
           $NewTabuList(m, TL, tlen), NewAspirationMap(m, AM)))$ 
  axiom  $\forall (x : D, curr : R, champ : R, nochamp : Nat, tlen : Nat, TL : TabuList,$ 
     $AM : AspirationMap, newnochamp : Nat, MS : NE-MSet, m : Move)$ 
     $(nochamp = (relax\ nonzero?)(succ(newnochamp)) \wedge \neg GOTest(x, curr)$ 
       $\wedge GetAdmissibleMoves(GetCandidateMoves(x, curr), TL, AM)$ 
         $= (relax\ nonempty-set?)(MS)$ 
       $\wedge m = ChooseMove(GetBestMoves(MS))$ 
       $\wedge new = (project\ 4)(m)$ 
       $\wedge Cost(x, champ) \leq Cost(x, new)$ 
       $\Rightarrow TabuStep(x, curr, champ, newnochamp, tlen, TL, AM)$ 
         $= TabuStep(x, new, champ, newnochamp, tlen,$ 
           $NewTabuList(m, TL, tlen), NewAspirationMap(m, AM)))$ 
end-definition
end-spec

```

Figure 128. Mediator Spec for Tabu Search with Aspiration, Cont.

Correctness of the program scheme is based on the following theorem.

Theorem 7.1.1 *In the spec TASProgram, TabuStep is well-defined and satisfies*

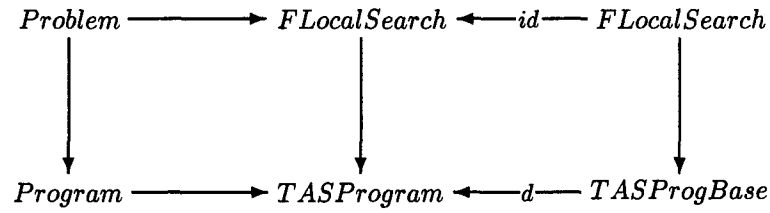
$$\begin{aligned} & \forall (x : D, curr : R, champ : R, nochamp : Nat, tlen : Nat, TL : TabuList, \\ & \quad AM : AspirationMap) \\ & (I(x) \wedge O(x, curr) \wedge O(x, champ) \\ & \quad \Rightarrow O(x, TabuStep(x, curr, champ, nochamp, tlen, TL, AM))) \end{aligned}$$

Proof. *TabuStep* is recursively defined using three conditional equations. The first proof obligation is that the three conditions cover all possible inputs. The condition in the first axiom is a disjunction of three terms. This condition is negated in the second and third axioms, and sometimes the negated condition is used to bind a variable. The second axiom adds a fourth condition comparing the costs of two solutions, and the third axiom negates this condition. Therefore the operation is defined for all inputs.

The second proof obligation is that the recursion terminates. First, if the input is invalid or if the current solution is not feasible with respect to this input, then *GetCandidateMoves* must return the empty set because by definition of *LegalMove* there are no elements of the sort *Move* that have an invalid input or an infeasible solution. Thus in this case the recursion terminates by returning the reigning champion.

Given a valid input and a feasible solution, if *nochamp* = zero or *GOTest* is true or no admissible moves exist, the recursion terminates. Otherwise a move is selected and made. If the move is strictly improving, *nochamp* is reset to its maximum value and the new solution replaces the champion. Otherwise *nochamp* is decremented by one. Since each champion is strictly better than the previous one, eventually the recursion must terminate.

Third is the condition to be satisfied. Assume that the antecedent holds. At each step of the search a move is made, which by feasibility of *N* is to a new feasible solution. This neighbor either



interpretation $\text{TabuSearch} : \text{Program} \Rightarrow \text{TASPProgBase}$ is
mediator TASPProgram
domain-to-mediator $\{F \rightarrow \text{TabuMain}\}$
codomain-to-mediator **import-morphism**

ip-scheme-morphism $\text{Feas-to-TS} : \text{FeasibleSolution} \rightarrow \text{TabuSearch}$ is
domain-sm $\{\}$
mediator-sm $\{\}$
codomain-sm $\{\}$

Figure 129. Program Scheme for Tabu Search with Aspiration

replaces the champion or it does not. When the recursion terminates, either the original champion is returned or a solution reachable from the original current solution is returned; in either case it is a feasible solution. $\square$

The correctness of TabuIter and TabuMain follows directly.

Figure 129 shows the whole program scheme as a refinement of the feasibility problem for FLocalSearch . The solution found by TabuMain will not in general be a local optimum, depending on the tabu rules, aspiration criteria and tunable parameters selected.

If $\text{MaxNoChamp}(x) = \text{zero}$, for example, TabuStep will return the champion passed to it and so TabuMain will return the best solution generated by InitSol . If $\text{MaxNoChamp}(x) = 1$, TabuStep will climb to a local optimum and most likely stop there, unless a single neutral or worsening move makes a new champion available. For larger values of $\text{MaxNoChamp}(x)$, the search for better local optima lasts longer. As long as aspiration by global objective is used, or another criterion that subsumes it, the tabu list will never prevent the champion from being a local optimum, though it may prevent visiting some local optima that would have been new champions. Thus a local optimum can be guaranteed under fairly broad conditions.

If $TabuLen(x) = zero$, the tabu list is empty, no moves are ever classified tabu and aspiration becomes irrelevant. Once a local optimum is reached, search will most likely wander around in close proximity to it until $MaxNoChamp$ is reached; in rare cases it may escape to a new local optimum before this happens, for example via neutral moves.

7.1.5 Example: Tabu Search for the Graph Partitioning Problem. Instantiating the program scheme for tabu search requires extending a refinement of *LocalSearch* by interpretation morphism to a refinement of *TASProgBase*, and composing the result with the interpretation morphism in Figure 129 that represents the tabu search program scheme, as explained in Chapter IV. As with the program schemes presented in Chapter VI, some of the extensions that *TASProgBase* adds to *LocalSearch* are well-defined, problem-independent components needed to support computation. The data structures added to *TASearch*—sequences of T , sets of *Move*, maps from A to A_{Level} —fall into this category. These can be added to a local search specification by the identity completion technique: that is, by pushout. Other components are problem-dependent design choices more or less unique to this particular program scheme, including tabu rules, aspiration criteria, and all the elements by which *TASProgBase* extends *TAS-Import*. These components can be defined before instantiation by extending the classification to provide definitions for them in problem domain terms, or can be deferred by using identity completion and providing definitions later in design. Either approach is correct, and they can be mixed. In Chapter VI it was suggested that all the design choices defined by *HillClimb*, *SimAnneal* and so forth be deferred. For tabu search, a mixed approach seems more natural.

To illustrate the process, the tabu search program scheme will be instantiated for the graph partitioning problem. First an additional classification step will be performed to identify a tabu rule and an aspiration criterion. These components are the essence of tabu search and influence later design choices. If multiple lists are used, for example, the choice of program scheme itself is affected. This classification step extends the local search specification for graph partitioning

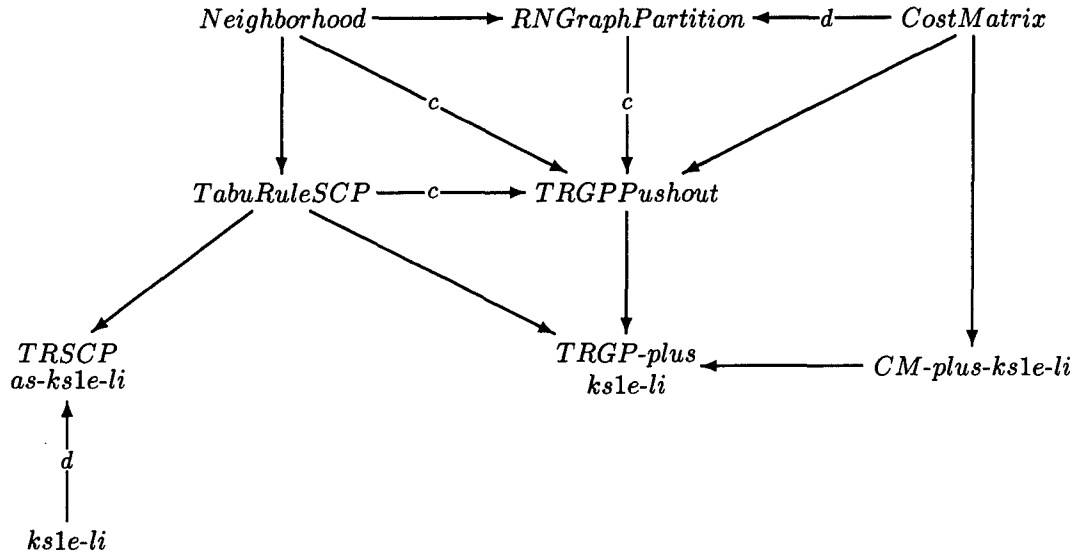
developed in Section 6.2.4 to a specification for tabu search with aspiration, that is, a refinement of *TASearch*. This refinement will then be extended by identity completion to a refinement of *TASProgBase* and composed with the tabu search program scheme given above. The elements that *TASProgBase* adds to *TAS-Import* will be treated as deferred design choices. The specs and refinements of the program scheme were developed with this sequence of steps in mind, but it would not be difficult for a designer to perform the steps in a different order.

7.1.5.1 A Tabu Rule for Graph Partitioning. In order to perform tabu search for the graph partitioning problem, we need to devise a tabu rule. There are many strategies to choose from; we shall pick lock-in. The neighborhood for graph partitioning is an adaptation of k -subset-1-exchange, and a specification for a lock-in rule for this neighborhood has already been given, so one might expect to be able to adapt the tabu rule to graph partitioning as well. This turns out to be more complicated than anticipated. The lock-in rule for k -subset-1-exchange cannot be used directly because graph partitioning is not an instance of a k -subset problem, nor is k -subset an instance of a graph partitioning problem. In general, and in reference to Figure 51, a connection from B to A with respect to some spec T neither constitutes nor implies a morphism or an interpretation from either B or A to the other. The connection shows that A and B are similar enough in their S structure to adapt the T -structure of B to A , but this does not guarantee that an arbitrary refinement of B can be incorporated meaningfully into A . To show that A and B are compatible with respect to an extension T' of T , the T -connection from B to A needs to be extended and verified.

Extending and verifying a connection is very similar to constructing one. The steps for extending the *Neighborhood* connection from k -subset-1-exchange to graph partitioning to incorporate the lock-in tabu rule are as follows:

1. The neighborhood specification for graph partitioning and the lock-in rule for k -subset-1-exchange are arranged as shown in Figure 130. Lock-in satisfies strong cycle prevention, so *TabuRuleSCP* is used in the role of T' .
2. As in previous examples, an initial program scheme $\text{TabuRuleSCP} \Rightarrow \text{CostMatrix}$ is formed by computing a pushout.
3. The interpretation scheme of step 2 is combined with the lock-in rule for k -subset-1-exchange. Since the neighborhood for graph partitioning was constructed via an earlier connection, the mediator spec *RNGraphPartition* contains a complete image of k -Subset-1-Exchange. If we combine *RNGraphPartition* with *TRSCP-as-ks1e-li* as a coproduct, the result will contain two images of it. Using a shape morphism, we can glue the specs on *ExtSet*, but this still leaves two copies of all the elements by which k -Subset-1-Exchange extends *ExtSet*, including much of the definition of the k -subset problem and the sorts and operations that define a neighborhood for it. The proper glue is k -Subset-1-Exchange itself, but the shape morphism mechanism is too weak to allow it. Figure 131 shows an alternative way to combine them. The interpretation *ks1e* is attached to node $N1$ in the diagram. The interpretation attached to node $N2$ is not based on *RNGraphPartitioning* but instead maps *Neighborhood* to the image of k -Subset-1-Exchange contained in *RNGraphPartition*. The interpretation morphism attached to $N1 \rightarrow N2$ likewise identifies this image in the mediator. This interpretation morphism is combined by colimit with the one constituting the tabu rule. The result has the lock-in extensions for k -subset-1-exchange in the same spec with the graph partitioning neighborhood so that the connection can be extended.

The colimit interpretation is a refinement of *TRGP0*, as shown by the interpretation *TRGP1*, in which we have returned our attention to the graph partitioning problem. *TRGP1* is *not* the interpretation at the bottom of the diagram in Figure 131.



spec *TRGPPushout* is
colimit of diagram

nodes *Neighborhood*, *TabuRuleSCP*, *RNGraphPartition*

arcs *Neighborhood* $\rightarrow$ *TabuRuleSCP* : { }

Neighborhood $\rightarrow$ *RNGraphPartition* :

{ *D* $\rightarrow$ *CostMatrix*, *I* $\rightarrow$ *GP-I*, *R* $\rightarrow$ *VSet*, *O* $\rightarrow$ *GP-O* }

end-diagram

ip-scheme *TRGP0* : *TabuRuleSCP* $\Rightarrow$ *CostMatrix* is

mediator *TRGPPushout*

domain-to-mediator cocone-morphism from *TabuRuleSCP*

codomain-to-mediator { }

ip-scheme-morphism *RNGP-to-TRGP0* : *RNGraphPartitioning* $\Rightarrow$ *TRGP0* is

domain-sm { }

mediator-sm cocone-morphism from *RNGraphPartition*

codomain-sm identity-morphism

Figure 130. Adapting Lock-In for *k*-Subset-1-Exchange to Graph Partitioning, Steps 1-3

4. The axiom of *TabuRuleSCP* is analyzed, yielding the following polarities:

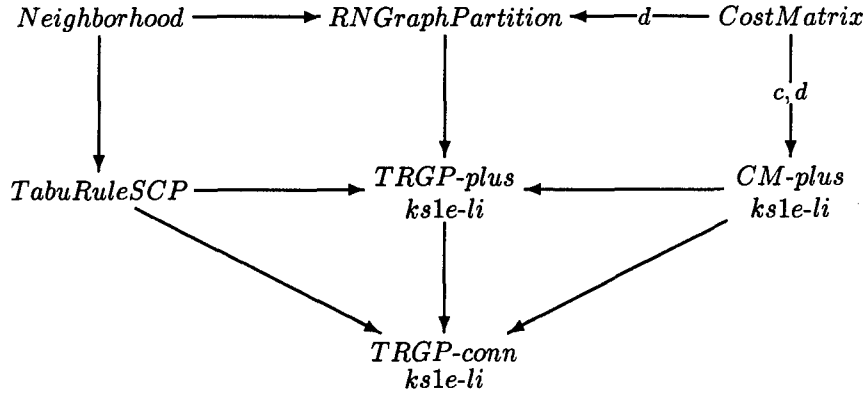
$$\begin{array}{rcl}
 \pi_D & = & - \\
 \pi_R & = & - \\
 \pi_C & = & - \\
 \pi_T & = & + \\
 \pi_{Move} & = & - \\
 \pi_{N^*} & = & - \\
 \pi_{LegalMove} & = & - \\
 \pi_{GetTabu} & = & \pm \\
 \pi_{HasTabu} & = & +
 \end{array}$$

The polarities for *C* and *N** are consistent with their definitions in the original connection. The polarities of *T*, *GetTabu* and *HasTabu* do not really matter, as they will be defined for graph partitioning in terms of *k*-subset as if they had neutral polarities. It is important that the polarities for *D* and *I* did not change, since that would mean the original connection was incompatible with the new axiom: any polarity other than $-$ would make the overall polarity neutral, requiring the sorts *CostMatrix* and *KS-D* to be isomorphic, which they are not, or requiring the respective input conditions to be equivalent, which they also are not.

The axiom defining *LegalMove* was not included in the analysis, as explained in Section 5.2.4. If it had been, it would have made the polarity of *I* neutral and spoiled the extension. As a defined operation, it will have a connection condition to satisfy. The polarity for *LegalMove* is derived from the polarity for *Move*. The sort *Move* is defined in *TabuRuleSCP*, and hence in the pushout, so a conversion operation is needed between *Move* as defined for *k*-subset and as defined for graph partitioning.

As a result of this analysis, *TRGP-plus-ks1e-li* is extended to *TRGP-conn-ks1e-li* with the operations, definitions and axiom shown in Figure 132.

5. In Figure 133 the connection spec is extended by *TRGraphPartition* with a definition for h_{Move} derived by considering the existing relations between *D*, *R* and *C* for *k*-subset and graph partitioning already established by the connection. The connection condition for *LegalMove* is verified as a corollary of the existing definition of h_D . As with previous connections, the final



```

spec TRGP-conn-ksle-li is
  import TRGP-plus-ksle-li
  op  $h_{Move} : Move \rightarrow KS-Move$ 
  axiom LegalMove-condition is
     $\forall (CM : CostMatrix, V : VSet, c : C, W : VSet)$ 
       $(LegalMove(CM, V, c, W) \Rightarrow KS-LegalMove(h_D(CM), V, c, W))$ 
  sort-axiom  $T = KS-T$ 
  definition of GetTabu is
    axiom  $\forall (m : Move) (GetTabu(m) = KS-GetTabu(h_{Move}(m)))$ 
  end-definition
  definition of HasTabu is
    axiom  $\forall (m : Move, t : T) HasTabu(m, t) \Leftrightarrow KS-HasTabu(h_{Move}(m), t)$ 
  end-definition
end-spec

interpretation TRGP2 : TabuRuleSCP  $\Rightarrow$  CM-plus-ksle-li is
  mediator TRGP-conn-ksle-li
  domain-to-mediator  $\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O\}$ 
  codomain-to-mediator  $\{\}$ 

ip-scheme-morphism RNGP-to-TRGP2 : RNGraphPartitioning  $\Rightarrow$  TRGP2 is
  domain-sm  $\{\}$ 
  mediator-sm  $\{\}$ 
  codomain-sm cocone-morphism from CostMatrix

```

Figure 132. Adapting Lock-In for k -Subset-1-Exchange to Graph Partitioning, Step 4

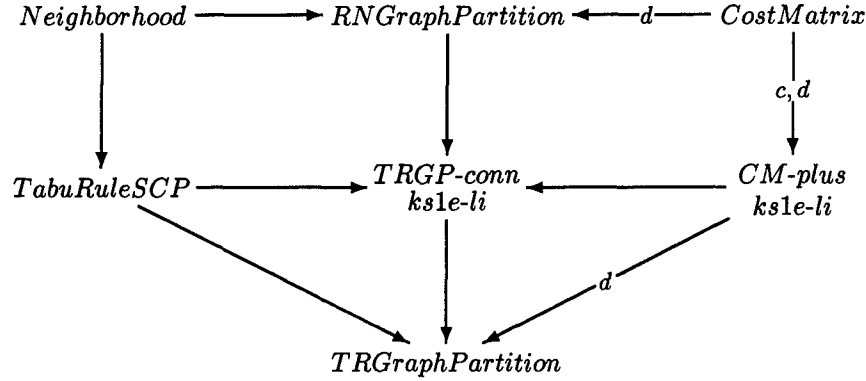
spec is shown importing a spec that contains both axioms that needed to be proved and the definitions derived to make those axioms theorems. Ideally the spec would be reconstructed to remove the axioms that have been satisfied.

Since the definitions for h_D , C , R and so on from the original connection are sufficient for the extensions of *TabuRuleSCP*, and since all symbols of *TabuRuleSCP* are now defined, we have a valid interpretation expressing how to implement a lock-in rule for graph partitioning.

The adaptation of lock-in for graph partitioning has been presented in a somewhat *ad hoc* fashion, so it is reasonable to ask how general the method just outlined is. The overall process followed is very similar to the connection mechanism described in Chapter V. There is additional verification involved, to make sure that the polarities of elements of T generated by analyzing axioms of T' are satisfied by the definitions derived for the original connection. Since many of these definitions are consistent with neutral polarity, this verification focuses once again on elements introduced in S . Constructing new definitions for the extensions in T' is handled as it was before.

The failure of the shape morphism mechanism to merge two interpretations in step 3 with sufficient sharing may indicate a weakness in the diagram refinement technique itself, or may be idiosyncratic to this application. The problem in general is that in a diagram of refinements there may be sharing among the mediators in addition to the sharing that a shape morphism identifies among the target specs. The global search connection for K -queens exhibited the same problem with respect to the spec *Nat*. Further, if two target specs share significant structure, it is not hard to imagine that they might be extended by the same definitions in two different refinements, so that these definitions in the mediators should also be identified. This problem should be studied further to identify a formal mechanism for mediator sharing.

In the context of extending a connection, the particular approach taken here seems generally applicable. As a result of building a connection previously, the final interpretation always contains a complete image of both $S \Rightarrow A$ and $T \Rightarrow B$, so the proper glue when extending the connection



```

spec TRGraphPartition is
  import TRGP-plus-ksle-li

  op  $h_{Move} : Move \rightarrow KS-Move$ 
  definition of  $h_{Move}$  is
    axiom  $\forall(m : Move, CM : CostMatrix, V : VSet, c : C, W : VSet, m' : KS-Move)$ 
       $((CM, V, c, W) = (\text{relax } LegalMove)(m)$ 
         $\wedge (h_D(CM), V, c, W) = (\text{relax } KS-LegalMove)(m'))$ 
         $\Rightarrow h_{Move}(m) = m')$ 
    end-definition

  theorem LegalMove-condition is
     $\forall(CM : CostMatrix, V : VSet, c : C, W : VSet)$ 
       $(LegalMove(CM, V, c, W) \Rightarrow KS-LegalMove(h_D(CM), V, c, W))$ 

  sort-axiom  $T = KS-T$ 

  definition of GetTabu is
    axiom  $\forall(m : Move) (GetTabu(m) = KS-GetTabu(h_{Move}(m)))$ 
  end-definition

  definition of HasTabu is
    axiom  $\forall(m : Move, t : T) HasTabu(m, t) \Leftrightarrow KS-HasTabu(h_{Move}(m), t)$ 
  end-definition
end-spec

interpretation TRGP : TabuRuleSCP  $\Rightarrow$  CM-plus-ksle-li is
  mediator TRGraphPartition
  domain-to-mediator  $\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O\}$ 
  codomain-to-mediator  $\{\}$ 

ip-scheme-morphism RNGP-to-TRGP : RNGraphPartitioning  $\Rightarrow$  TRGP is
  domain-sm  $\{\}$ 
  mediator-sm  $\{\}$ 
  codomain-sm cocone-morphism from CostMatrix

```

Figure 133. Adapting Lock-In for k -Subset-1-Exchange to Graph Partitioning, Step 5

to $T' \Rightarrow B'$ is T -as- B for the mediators and B in the target. By focusing on $T \Rightarrow B$ and ignoring $S \Rightarrow A$ temporarily, the proper combination can be carried out by diagram refinement. If B is not the only sharing between $A + B$ and B' , this refinement can include a shape morphism. The focus is then changed back to $S \Rightarrow A$ by composing morphisms to obtain an interpretation morphism from $T \Rightarrow A + B$ to the colimit refinement.

The graph partitioning example raises some issues that were somewhat glossed over above concerning the details of polarity analysis as applied to higher-order or polymorphic operations, such as the relax operation. Relax is applied to *LegalMove* itself, not to a value it returns. Should this affect the polarity of *LegalMove*, and if so, how? The relaxation is applied to a move and compared with another value, but clearly we do not want to generate a neutral polarity. An equality that serves only to bind a variable should perhaps be treated differently in polarity analysis than equalities that actually constrain behavior. Issues like this need to be studied further, with the goal of generating polarities that are as weak as possible and still guarantee correctness.

It is interesting to observe that the extension for lock-in used *RNGraphPartitioning* as its starting point rather than *PSGraphPartitioning*. The latter would not have worked because *PSGraphPartitioning* was “cleaned up” so that it no longer contained an image of k -subset-1-exchange. Clean up of a connection to remove what seem to be unnecessary layers of definition and unused symbols is not a good idea if one anticipates using refinements of $T \Rightarrow B$ after the connection is complete.

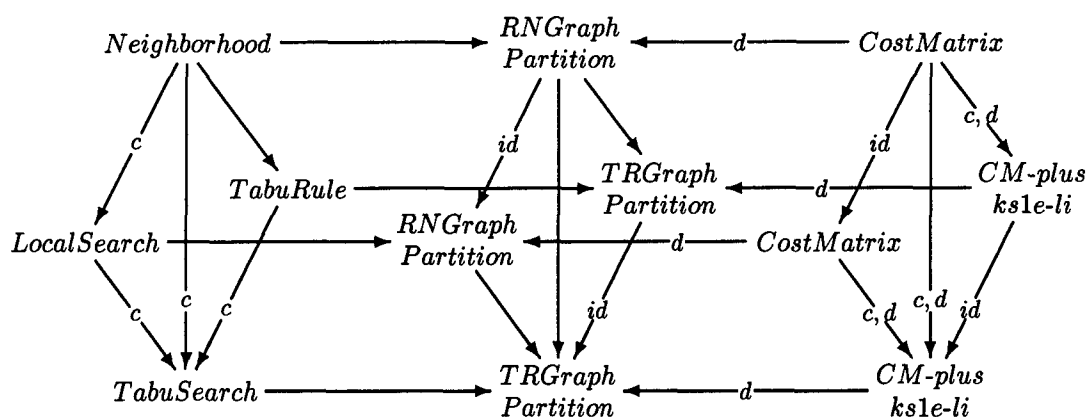
Overall it was surprising that such heavy machinery was required to make use of a refinement of k -subset-1-exchange. In retrospect it is clear that a connection must be extended with just as much care as went into its construction, but intuitively one also expects specialized knowledge to be easier to apply than generic knowledge. Since graph partitioning was already known to have a neighborhood of a particular kind, for example, adapting lock-in to it was expected to be quite simple. On the other hand, by this time the strictness of formal methods should surprise no one.

If, during the adaptation of a neighborhood to a problem, a non-trivial *FC* is used, the problem of adapting a tabu rule becomes more difficult. The neighborhood tactic does not produce a connection, so the technique above does not apply directly. *FC* strengthens *N_info_A*, which is consistent with the negative polarity of *N_info* with respect to the feasibility axiom, but not the positive polarity required by reachability. Moreover, the connection condition between *O_A* and *O_B* is never established, though since the polarity analysis of *TabuRuleSCP* does not yield a polarity for *O*, this may not matter. Intuitively, if a neighborhood has been strengthened, tabu rules should still work. Rules that depend on certain properties such as symmetry would require that these properties be reverified when *FC* is introduced. Symmetry, for example, is not affected by *FC* for any neighborhood. It seems plausible, then, that something reasonable can be done for this case. This issue remains open as a topic for future research.

7.1.5.2 Completing the Instantiation. Now that we have a tabu rule for graph partitioning, we will add aspiration by global objective. Figure 134 shows a diagram refinement that adds the tabu rule to the full specification of graph partitioning to get a refinement of *TabuSearch*. The specs *TRGraphPartition* and *CM-plus-ks1e-li* already contain all they need, so this refinement does not create any new specs.

The domain theory for aspiration by global objective, *GlobObj*, is a definitional extension of *TabuSearch*, so it can be incorporated into graph partitioning by a pushout. This is composed with the interpretation *AspByGlobObj* to give a refinement of *TASearch*. These two steps are shown in Figure 135.

Figure 136 shows the final stages of instantiation. The refinement of *TASearch* is extended to *TASProgBase* by pushout, adding data structures and deferring the design choices contained therein. The target spec needs the same extensions, but there is no convenient way to construct it. Instead of showing all the diagrams and colimits involved, *TASPCM-ks1e-li* is simply described as being *TASPBGraphPartition* with certain definitions removed. The last step, not shown, is to



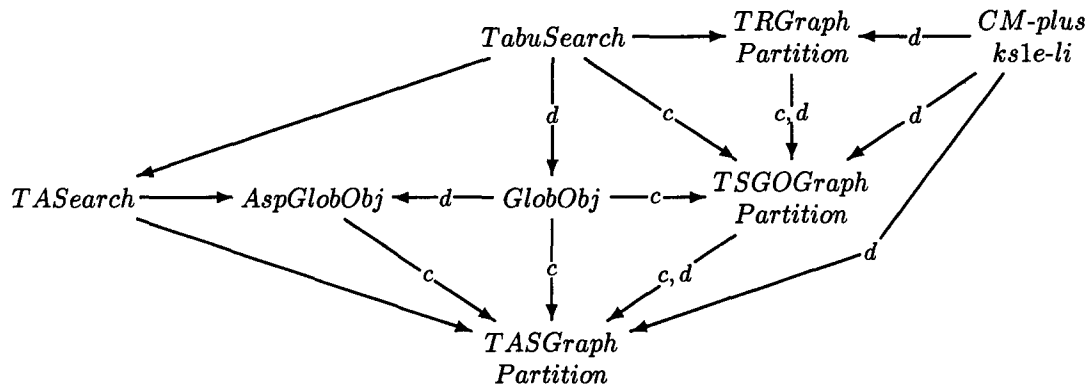
interpretation $TSGP : TabuSearch \Rightarrow CM-plus-ks1e-li$
mediator $TRGraphPartition$
domain-to-mediator $\{D \rightarrow CostMatrix, I \rightarrow GP-I, R \rightarrow VSet, O \rightarrow GP-O\}$
codomain-to-mediator $\{\}$

Figure 134. Tabu Search for Graph Partitioning

compose the two interpretation morphisms in parallel. Work can now begin on defining how initial solutions and candidate moves are found, tuning parameters, and so on.

7.2 The Kernighan-Lin Heuristic

The Kernighan-Lin heuristic for graph partitioning was described briefly in Chapter III. This algorithm searches through a solution space by making variable-length compound moves that effectively search a much larger sample of the space at each step than just the immediate neighborhood of the current solution. In light of the preceding discussion of tabu search, the Kernighan-Lin heuristic can be viewed as initializing an empty tabu list and then executing the lock-in strategy until no permissible moves remain, without using any aspiration criteria. For a graph with $2n$ nodes, exhaustion occurs after n moves, as each move removes two nodes from further consideration. This chain of moves is then examined and the best solution is selected as the new current solution. Search stops when this new solution is no better than the solution that was current when the step began. If search continues, the tabu list is re-initialized to empty. The actual Kernighan-



spec *TSGOGraphPartition* **is**

colimit of diagram

nodes *TabuSearch*, *TRGraphPartition*, *GlobObj*

arcs *TabuSearch* $\rightarrow$ *TRGraphPartition* :

$\{D \rightarrow \text{CostMatrix}, I \rightarrow \text{GP-I}, R \rightarrow \text{VSet}, O \rightarrow \text{GP-O}\},$

TabuSearch $\rightarrow$ *GlobObj* : **import-morphism**

end-diagram

spec *TASGraphPartition* **is**

colimit of diagram

nodes *GlobObj*, *AspGlobObj*, *TSGOGraphPartition*

arcs *GlobObj* $\rightarrow$ *AspGlobObj* : **import-morphism**,

GlobObj $\rightarrow$ *TSGOGraphPartition* : **cocone-morphism from** *GlobObj*

end-diagram

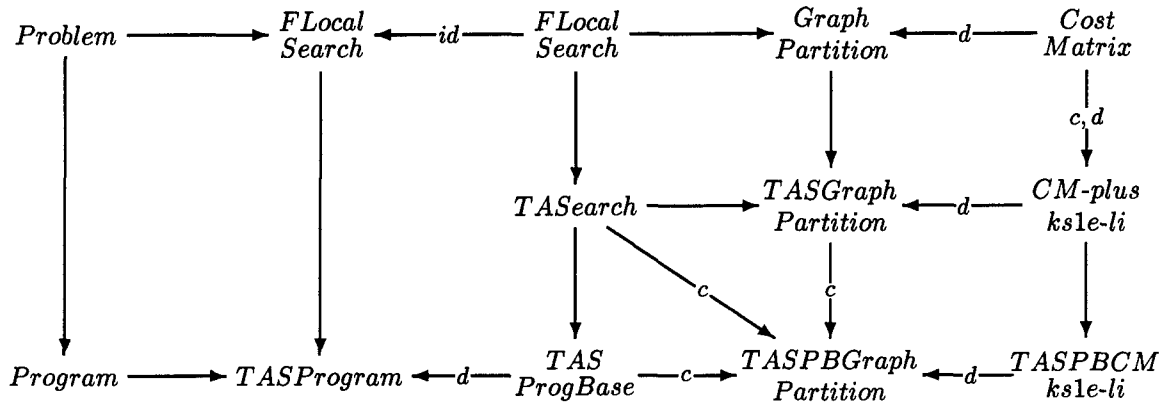
interpretation *TASGP* : *TASearch* $\Rightarrow$ *CM-plus-ks1e-li* **is**

mediator *TASGraphPartition*

domain-to-mediator $\{D \rightarrow \text{CostMatrix}, I \rightarrow \text{GP-I}, R \rightarrow \text{VSet}, O \rightarrow \text{GP-O}\}$

codomain-to-mediator $\{\}$

Figure 135. Instantiating Aspiration by Global Objective for Graph Partitioning



spec *TASPBGPartition* **is**
colimit of diagram
nodes *TASearch*, *TASGraphPartition*, *TASProgBase*
arcs *TASearch* $\rightarrow$ *TASGraphPartition* :
 $\{D \rightarrow \text{CostMatrix}, I \rightarrow \text{GP-I}, R \rightarrow \text{VSet}, O \rightarrow \text{GP-O}\},$
TASearch $\rightarrow$ *TASProgBase* : $\{\}$
end-diagram

spec *TASPBCM-ks1e-li* **is**
 $\langle$ Remove from *TASPBGPartition* those definitions
that *TASGraphPartition* adds to *CM-plus-ks1e-li* $\rangle$
end-spec

interpretation *TASBPBG* : *TASProgBase* $\Rightarrow$ *TASPBCM-ks1e-li* **is**
mediator *TASPBGPartition*
domain-to-mediator cocone-morphism from *TASProgBase*
codomain-to-mediator $\{\}$

Figure 136. Final Instantiation of Tabu Search for Graph Partitioning

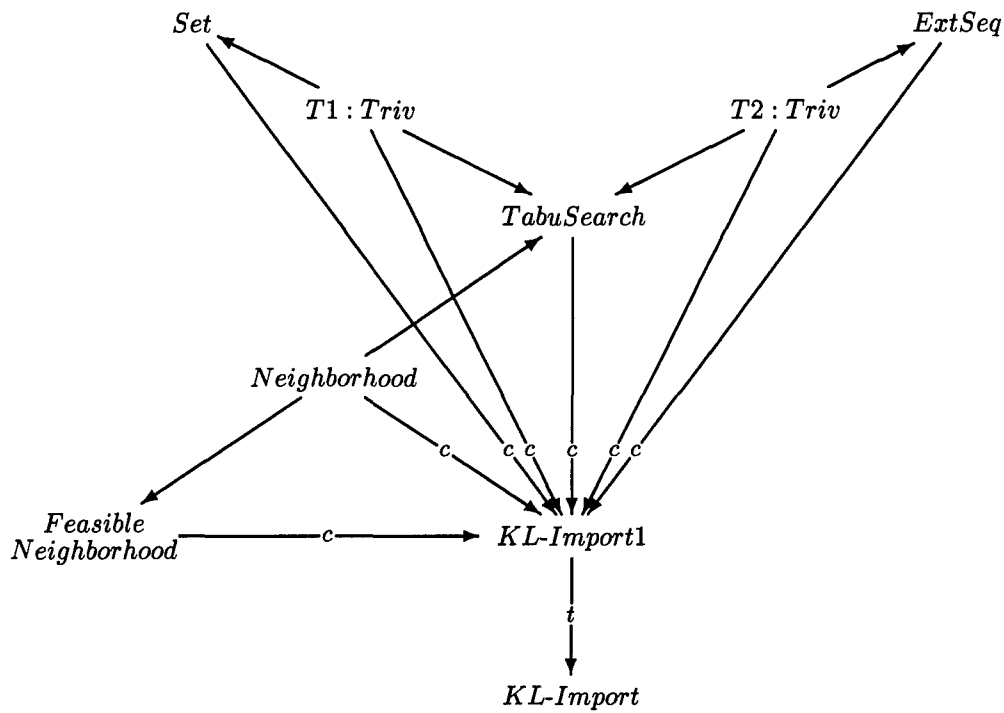
Lin algorithm is heavily optimized and for example does not use an explicit tabu list, but this characterization accurately captures its essence.

7.2.1 Program Scheme for Kernighan-Lin. Using the formalisms developed for tabu search in the previous section, it is possible to generalize the Kernighan-Lin heuristic to any local search problem for which a tabu rule can be defined. Figures 137 and 138 show a program scheme for Kernighan-Lin search. In keeping with the classic graph partitioning implementation, the theory includes a tabu rule but not aspiration criteria, though there is nothing preventing us from developing a variant that does use aspiration.

The domain theory spec, *Kernighan-Lin*, imports a spec that extends *TabuSearch* with the necessary data structures, namely sets of moves and sequences of tabu elements; a feasible neighborhood is also required. (If aspiration criteria are desired, *TAS-Import* could be used directly.) Standard extensions compute initial solutions and test for global optimality. *GetCandidateMoves* returns a subset of the neighborhood of the current solution, though typically the entire neighborhood is used. *ChooseMove* breaks ties among equally desirable moves. *ChooseChamp* breaks ties among equally desirable solutions in a chain of moves. The program scheme guarantees that a feasible solution will be found.

Figures 139 and 140 show the mediator spec of the program scheme. Many of its elements are variants of elements defined in the tabu search program scheme presented in Section 7.1.4. *NewTabuList* extends a tabu list without concern for its length. *IsTabu* checks whether a move is classified tabu by a tabu element. *GetAdmissibleMoves* culls tabu moves from a set of candidates; the aspiration check has been removed. *GetBestMoves* extracts the best moves from a set of moves. *NewChamp* chooses the better of two solutions found in a chain of moves, breaking ties with *ChooseChamp*. *KLMain* and *KLIter* initiate and perform a series of *MaxTries*(*x*) searches.

The heart of the program scheme is *KLStep* and *KLChain*. At each step of the search, *KLStep* calls *KLChain* to build a chain of moves. The chain runs until *GOTest* is true, no moves



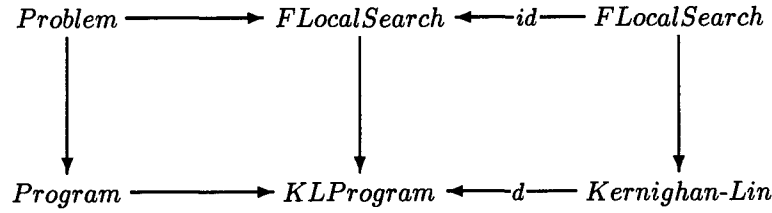
```

spec KL-Import1 is
  colimit of diagram
    nodes TabuSearch, T1 : Triv, Set, T2 : Triv, ExtSeq,
      Neighborhood, FeasibleNeighborhood
    arcs  T1 → TabuSearch : {E → Move}, T1 → Set : {}
      T2 → TabuSearch : {E → T}, T2 → ExtSeq : {}
      Neighborhood → TabuSearch : {}
      Neighborhood → FeasibleNeighborhood : import-morphism
  end-diagram

spec KL-Import is
  translate KL-Import1 by
    {Move → Move, Set → MSet, NE-Set → NE-MSet, T → T, Seq → TabuList}

```

Figure 137. Domain Theory for Kernighan-Lin Search



```

spec Kernighan-Lin is
  import KL-Import

  sort ISP

  op InitISP : D → ISP

  op InitSol : D, ISP → R, ISP
  axiom  $\forall(x : D, p : ISP) (I(x) \Rightarrow O(x, (\text{project } 1)(\text{InitSol}(x, p))))$ 

  op MaxTries : D → Pos

  op MaxChain : D → Pos

  op Optimal : D, R → Boolean
  definition of Optimal is
    axiom  $\forall(x : D, z : R) (Optimal(x, z) \Leftrightarrow$ 
       $\forall(z' : R) (O(x, z') \Rightarrow Cost(x, z) \leq Cost(x, z')))$ 
  end-definition

  op GOTest : D, R → Boolean
  axiom  $\forall(x : D, z : R) (GOTest(x, z) \Rightarrow Optimal(x, z))$ 

  op GetCandidateMoves : D, R → MSet
  axiom  $\forall(x : D, z : R, m : Move)$ 
     $(in(m, GetCandidateMoves(x, z))$ 
       $\Rightarrow x = (\text{project } 1)(m) \wedge z = (\text{project } 2)(m))$ 

  op ChooseMove : NE-MSet → Move
  axiom  $\forall(MS : NE-MSet) in(ChooseMove(MS), MS)$ 

  op ChooseChamp : D, R, R → R
  axiom  $\forall(x : D, z1 : R, z2 : R)$ 
     $(z1 = ChooseChamp(x, z1, z2) \vee z2 = ChooseChamp(x, z1, z2))$ 
end-spec

interpretation KLSearch : Program  $\Rightarrow$  Kernighan-Lin is
  mediator KLProgram
  domain-to-mediator  $\{F \rightarrow KLMain\}$ 
  codomain-to-mediator import-morphism

ip-scheme-morphism Feas-to-KL : FeasibleSolution → KLSearch is
  domain-sm {} mediator-sm {} codomain-sm {}
  
```

Figure 138. Program Scheme for Kernighan-Lin Search

```

spec KLProgram is
  import KL-Import

  op NewTabuList : Move, TabuList → TabuList
  definition of NewTabuList is
    axiom  $\forall(m : \text{Move}, TL : \text{TabuList}) (NewTabuList(m, TL) = \text{prepend}(\text{GetTabu}(m), TL))$ 
  end-definition

  op IsTabu : Move, TabuList → Boolean
  definition of IsTabu is
    axiom  $\forall(m : \text{Move}) \neg IsTabu(m, \text{empty-seq})$ 
    axiom  $\forall(m : \text{Move}, t : T, TL : \text{TabuList})$ 
       $(IsTabu(m, \text{prepend}(t, TL)) \Leftrightarrow HasTabu(m, t) \vee IsTabu(m, TL))$ 
  end-definition

  op GetAdmissibleMoves : MSet, TabuList → MSet
  definition of GetAdmissibleMoves is
    axiom  $\forall(TL : \text{TabuList})$ 
       $(GetAdmissibleMoves(\text{empty-set}, TL) = \text{empty-set})$ 
    axiom  $\forall(m : \text{Move}, MS : \text{MSet}, TL : \text{TabuList})$ 
       $(\neg IsTabu(m, TL)$ 
         $\Rightarrow GetAdmissibleMoves(\text{insert}(m, MS), TL)$ 
         $= \text{insert}(m, GetAdmissibleMoves(MS, TL)))$ 
    axiom  $\forall(m : \text{Move}, MS : \text{MSet}, TL : \text{TabuList})$ 
       $(IsTabu(m, TL)$ 
         $\Rightarrow GetAdmissibleMoves(\text{insert}(m, MS), TL)$ 
         $= GetAdmissibleMoves(MS, TL))$ 
  end-definition

  op GetBestMoves : NE-MSet → NE-MSet
  definition of GetBestMoves is
    axiom  $\forall(m : \text{Move}, MS : \text{NE-Set}) (in(m, GetBestMoves(MS)) \Leftrightarrow$ 
       $(in(m, MS) \wedge \forall(x : D, z : R, x' : D, z' : R, n : \text{Move})$ 
         $(x = (\text{project } 1)(m) \wedge z = (\text{project } 4)(m) \wedge in(n, MS)$ 
         $\wedge x' = (\text{project } 1)(n) \wedge z' = (\text{project } 4)(n)$ 
         $\Rightarrow Cost(x, z) \leq Cost(x', z'))))$ 
  end-definition

  op NewChamp : D, R, R → R
  definition of NewChamp is
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z1) < Cost(x, z2) \Rightarrow NewChamp(x, z1, z2) = z1)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow NewChamp(x, z1, z2) = z2)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R)$ 
       $(Cost(x, z2) = Cost(x, z1) \Rightarrow NewChamp(x, z1, z2) = ChooseChamp(x, z1, z2))$ 
  end-definition

  op BetterSol : D, R, R → R
  definition of BetterSol is
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z1) \leq Cost(x, z2) \Rightarrow BetterSol(x, z1, z2) = z1)$ 
    axiom  $\forall(x : D, z1 : R, z2 : R) (Cost(x, z2) < Cost(x, z1) \Rightarrow BetterSol(x, z1, z2) = z2)$ 
  end-definition

```

Figure 139. Mediator Spec for Kernighan-Lin Search

```

op KLMain : D → R
definition of KLMain is
  axiom  $\forall(x : D)$ 
     $(KLMain(x) = KLIter(x, MaxTries(x), MaxChain(x), InitISP(x)))$ 
end-definition

op KLIter : D, Pos, Pos, ISP → R
definition of KLIter is
  axiom  $\forall(x : D, max : Pos, p : ISP, z : R)$ 
     $(z = (\text{project } 1)(InitSol(x, p))$ 
       $\Rightarrow KLIter(x, succ(zero), max, p) = KLStep(x, z, max))$ 
  axiom  $\forall(x : D, k : Pos, max : Pos, p : ISP, z : R, p' : ISP, z' : R, z'' : R)$ 
     $(\langle z, p' \rangle = InitSol(x, p)$ 
       $\wedge z' = KLStep(x, z, max)$ 
       $\wedge z'' = KLIter(x, k, max, p')$ 
       $\Rightarrow KLIter(x, succ(nat-of-pos(k)), max, p) = BetterSol(x, z', z''))$ 
end-definition

op KLStep : D, R, Pos → R
definition of KLStep is
  axiom  $\forall(x : D, z : R, max : Pos, z' : R)$ 
     $(GOTest(x, z) \vee (z' = KLChain(x, z, z, nat-of-pos(max), empty-seq)$ 
       $\wedge Cost(x, z) \leq Cost(x, z'))$ 
       $\Rightarrow KLStep(x, z, max) = z)$ 
  axiom  $\forall(x : D, z : R, max : Pos, z' : R)$ 
     $(\neg GOTest(x, z) \wedge z' = KLChain(x, z, z, nat-of-pos(max), empty-seq)$ 
       $\wedge Cost(x, z') < Cost(x, z)$ 
       $\Rightarrow KLStep(x, z, max) = KLStep(x, z', max))$ 
end-definition

op KLChain : D, R, R, Nat, TabuList → R
definition of KLChain is
  axiom  $\forall(x : D, curr : R, champ : R, max : Nat, TL : TabuList)$ 
     $(max = zero \vee GOTest(x, curr)$ 
       $\vee GetAdmissibleMoves(GetCandidateMoves(x, curr), TL)$ 
       $= empty-set$ 
       $\Rightarrow KLChain(x, curr, champ, max, TL) = champ)$ 
  axiom  $\forall(x : D, curr : R, champ : R, max : Nat, TL : TabuList, newmax : Nat,$ 
     $MS : NE-MSet, m : Move, new : R)$ 
     $(max = nat-of-pos(succ(newmax)) \wedge \neg GOTest(x, curr)$ 
       $\wedge GetAdmissibleMoves(GetCandidateMoves(x, curr), TL)$ 
       $= (\text{relax nonempty-set?})(MS)$ 
       $\wedge m = ChooseMove(GetBestMoves(MS)) \wedge new = (\text{project } 4)(m)$ 
       $\Rightarrow KLChain(x, curr, champ, max, TL)$ 
       $= KLChain(x, new, NewChamp(x, champ, new), newmax,$ 
       $NewTabuList(m, TL)))$ 
end-definition
end-spec

```

Figure 140. Mediator Spec for Kernighan-Lin Search, Cont.

are admissible, or the limit on chain length is reached. The best solution found along the chain is returned by *KLChain*. If this solution is better than the current solution (i.e., the first solution in the chain), the process repeats from the new solution. If not, the current solution is returned. That the recursion of *KLChain* terminates follows by induction on the natural numbers: *max* is reduced by one at each iteration and must eventually reach zero. Since *KLStep* calls *KLChain* with an empty tabu list, for the first link the entire set of candidate moves is evaluated. If the entire chain fails to improve on the current solution, then the current solution must be as good as the best moves available for the first link. If *GetCandidateMoves* returns the whole neighborhood, then failure of *KLChain* to find a better solution implies local optimality of the current one. Since we allow restricted candidate sets, however, we cannot make this claim in general.

If $MaxChain(x) = 1$, each step makes the best available move from the current solution, a chain of length one, and search stops when this move is not improving; this is equivalent to steepest ascent hill climbing. For $MaxChain(x) > 1$, each step looks deeper into the territory surrounding the current solution. If the solution space is finite and each solution can be transformed by a finite number of transform values, the chain will always terminate by running out of non-tabu moves, allowing *MaxChain* to be dispensed with, if desired.

7.2.2 Experiments with Neighborhoods. There is little published information on the strength or effectiveness of different tabu rules. Occasionally a paper will study more than one neighborhood for a problem (1, 12, 81), but we are unaware of any systematic studies of tabu rules themselves. Section 7.1.2 provides qualitative information on the relative strengths of several rules for some particular neighborhoods. In order to understand more quantitatively the nature and behavior of these rules, experiments were run in which the tabu list could grow without bound and moves were made randomly until a solution was reached from which all moves were tabu, at which point we say the run is *exhausted* or has reached exhaustion. The results have special relevance to

the Kernighan-Lin heuristic but also provide empirical guidance for regular tabu search in choosing appropriate tabu rules and especially appropriate tenures for maintaining tabu status.

Two neighborhoods were the focus of the study: k -subset-1-exchange and array-swap-index. All of the experiments look at some variant of a lock-out strategy, and the primary statistic gathered is the *run length*, or how many moves were made before all were tabu. The run length varies with the exact sequence of moves, so multiple runs were made for each case examined. Run times were also gathered but depend strongly on the implementation of the test software and on the hardware platform used, so these results are not reported. Lock-in rules were not studied experimentally because they tend to be very easy to analyze directly. Analytical results are compared below with the experimental results at appropriate points.

7.2.2.1 k -Subset-1-Exchange. Section 7.1.2.2 describes five tabu rules for the k -subset-1-exchange neighborhood and relates them to each other in Figure 110. The most restrictive rule is lock-in. If n is the size of the set S and k the size of the subset desired, then the lock-in rule will reach exhaustion in either k or $(n - k)$ moves, whichever is less. The single-attribute rule that forbids an element added to the current solution from being dropped will reach exhaustion in k moves. The other single-attribute move, forbidding an element that has been dropped from being added back, reaches exhaustion in $(n - k)$ moves. Each of these rules reaches exhaustion in exactly the number of moves indicated, with no variation.

If both single-attribute rules are applied via free conjunction, the situation is more complex. The minimum run length is the lesser of k and $(n - k)$, as for regular lock-in, and the maximum run length is one more than the larger of k and $(n - k)$. Let $T1$ be the tabu list of elements that must be in the current solution, and let $T2$ be the tabu list of elements that must be outside the current solution. When a move $\langle a, b \rangle$ is made, a is added to $T1$ and b is added to $T2$. After k moves, $T1$ contains k elements. If the current solution contains exactly these elements, $T1$ is not violated. On all subsequent moves, however, $T1$ contains more than k elements and so must be

violated. By similar logic, once more than $(n - k)$ moves have been made, $T2$ must be violated. The maximum run length is therefore the larger of k and $(n - k)$, plus one to insure both lists are violated.

Now assume without loss of generality that k is less than or equal to $(n - k)$. The minimum run length occurs when after k moves the current solution contains all of the elements of $T1$ and at least two elements of $T2$ are violated. No moves remain permissible, since all moves violate $T1$ and no single move can fix two or more violated elements of $T2$. For example, if $k = 3$ and $n > 6$, the sequence of moves

$$\langle d, a \rangle \langle a, b \rangle \langle b, c \rangle$$

leaves no legal moves remaining. The current solution is $\{a, b, d\}$, $T1 = [d, a, b]$ and $T2 = [a, b, c]$. Since $T1$ is not violated, this solution is legal even though it violates two elements of $T2$. All moves from it yield solutions that violate both $T1$ and $T2$, so the run is exhausted.

Table 1 shows the results achieved by 45 random trials for each of several values for n and k . It lists the actual run lengths that were observed. All were within one of either the minimum or maximum value. For the first few moves of a run neither tabu list is violated. Eventually one of them is violated, so all subsequent moves must avoid violating the other. Whichever list is violated first, then, tends to determine which value the run length will take. It is possible to switch lists if the violated list has only one violated element and a move corrects it, but the effect of switching is minimal. Intermediate run lengths between the minimum and maximum are either not possible or rarely seen.

The rule that combines the attributes in correspondence is the lock-out rule, and it is even more difficult to analyze. The rule is less restrictive than free conjunction since a pair of elements must be violated together; hence multiple attributes of each kind can be violated as long as they do not occur together on the tabu list. Empirical studies suggest that the maximum run length is bounded by $2n$. Figure 141 shows maximum run lengths for multiple random trials with $k = 3$

n	k	run length	n	k	run length
12	3	3, 4, 9, 10	48	3	3, 45, 46
	4	4, 8, 9		12	12, 36, 37
	6	6, 7		16	16, 17, 32, 33
				24	24, 25
24	3	3, 21, 22	60	3	3, 57, 58
	6	6, 7, 18, 19		15	15, 16, 45, 46
	8	8, 9, 16, 17		20	20, 21, 40, 41
	12	12, 13		30	30, 31
36	3	3, 33, 34	72	3	3, 69, 70
	9	9, 10, 27, 28		18	18, 19, 54, 55
	12	12,13,24, 25		24	24, 25, 48, 49
	18	18, 19		36	36, 37

Table 1. Observed Run Lengths for k -Subset-1-Exchange with Free Conjunction

over several values of n . The data is very well approximated by a line with a slope slightly in excess of one. If moves are not chosen randomly, longer runs are possible. A technique that orders the elements of the set and always makes the legal move involving the smallest elements is linear with a slope of $7/4$. Such runs must be exceedingly rare since the random runs did not do nearly so well. The ordering technique for values of k larger than 3 approaches $2n$ but never reaches it, and random runs for large k do as well or better, but still never exceed $2n$. Figure 142 shows maximum run lengths where $k = n/2$ rather than being fixed. The data is again linear, but with a slope much closer to 2. Maximum run lengths for $k = n/3$ and $k = n/4$ are similar, with slopes of 1.72 and 1.62, respectively. Minimum-length observed runs, not shown, are about 75% as long as maximum-length runs.

All of these tabu rules for k -subset-1-exchange reach exhaustion after runs linearly proportional to the size of the input set, so any of them might be a good choice in an algorithm. The classic application to graph partitioning uses lock-in, so runs (or chains) are $k = n/2$ moves long. The discussion above indicates some alternatives that allow longer chains and hence may find better local optima. Whether any improvement justifies the added effort is a topic for empirical studies. For general tabu, the results support the intuition that dropped nodes be treated differently from added nodes by giving them different tenures based on the values of k and $(n - k)$.

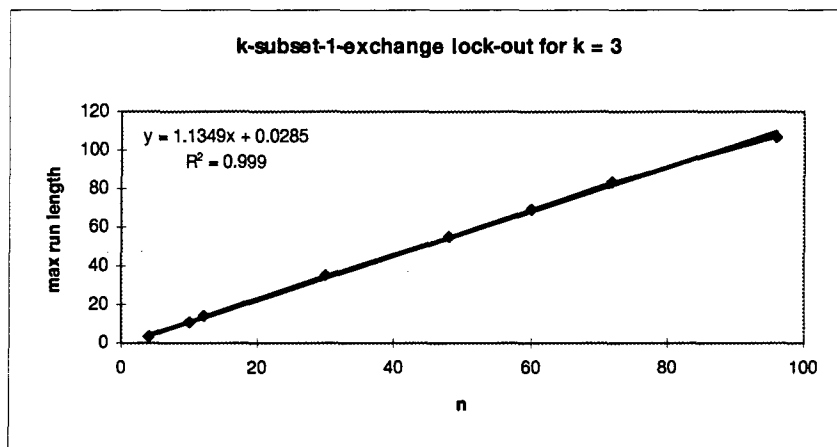


Figure 141. Maximum Run Length for k -Subset-1-Exchange with Lock-Out, $k = 3$

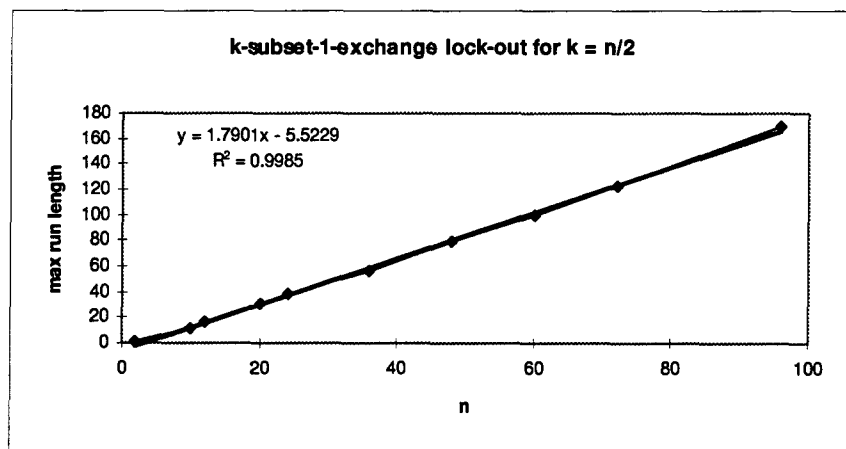


Figure 142. Maximum Run Length for k -Subset-1-Exchange with Lock-Out, $k = n/2$

7.2.2.2 *Array Swap Index.* Section 7.1.2.3 describes a large number of tabu rules for the array-swap-index neighborhood; Figure 113 indicates relations among them. The lock-in rules on the right of the figure are for the most part easy to analyze. Let n be the size of an array. The straight lock-in rule fixes two elements in place at each move, exhausting a run in $n/2$ moves. The single-attribute rules lock in one of the two elements. In most cases there is no difference between fixing the one at i and fixing the one at j . This rule reaches exhaustion in $n - 1$ moves (the array element not fixed at the end of the run has no other element with which to swap). If both single-attribute rules are combined via free conjunction, exhaustion is still reached in $n - 1$ moves. The proof is analogous to the one for k -subset-1-exchange: by the time $T1$ and $T2$ each have $n - 1$ entries, they each attempt to fix all elements but one in place, so neither permits any moves. Prior to that, either list may be heavily violated as long as the other is not, and as long as such a list is shorter than $n - 1$, moves are possible. Finally, the rule that combines both single attributes by correspondence also reaches exhaustion in $n - 1$ moves. This was discovered experimentally; the reasons are not entirely clear. This rule is less restrictive than the others, so $n - 1$ is a lower bound. Proving that it is also an upper bound will be left for future research.

Even though these lock-in rules—single-attribute, free conjunction and correspondence—reach exhaustion in the same number of steps, they do have subtle differences. These are best seen by considering how many moves are legal at each step of a chain. For the single-attribute rule, each move fixes one element, making the rest of the solution act like an array that is one element shorter. For $n = 25$, for example, the number of legal moves evolves according to the sequence

300, 276, 253, 231, 210, 190, 171, 153, 136, 120,
105, 91, 78, 66, 55, 45, 36, 28, 21, 15, 10, 6, 3, 1

because an array of length n has $\frac{1}{2}n(n - 1)$ possible moves. If separate attributes are combined by free conjunction, the exact numbers vary with the choice of moves but a typical sequence is

300, 299, 296, 291, 284, 275, 264, 153, 136, 120,
105, 91, 78, 66, 55, 45, 36, 28, 21, 15, 10, 6, 3, 1

Here the early moves are less constrained: every move permitted by the single-attribute rule is allowed, as well as moves that violate an element on one tabu list but not the other. Once enough moves have been made for violations to be likely or required for continuation, the sequence matches the single-attribute case for the rest of the run. The correspondence rule is even less restrained. A typical sequence is

300, 299, 298, 252, 251, 250, 208, 207, 206, 205, 204,
184, 165, 131, 115, 100, 86, 62, 51, 33, 25, 13, 5, 2

Here at every step of the chain the correspondence rule offers more freedom of choice than the other rules, and a much slower decline early in the chain. This can have a significant effect on a tabu search. In particular, the correspondence rule can support longer tenures than would be wise for the others.

Lock-out rules for array-swap-index were studied experimentally; they are difficult to approach analytically. Mean run lengths for the standard lock-out rule are shown in Figure 143 for a range of array sizes. The runs are long: proportional to the cube of n . For Kernighan-Lin, it takes too long to run a chain all the way to exhaustion with the lock-out rule. For tabu search, this result means that tabu lists can be quite long without overly constraining the search. For this rule and the other lock-out rules, the variance between minimum and maximum run lengths was small: minimum run lengths were typically about 85% of the maximums. Distributions were roughly bell shaped, skewed to the right some.

Figure 144 shows separate attributes combined with free conjunction. The mean run length has a slower growth rate, quadratic rather than cubic. Figure 145 shows the single-attribute rule. The growth rate is virtually the same. The curve fitting process derived a higher coefficient for the quadratic term for the single-attribute rule, but in the actual data free conjunction had a slightly greater mean run length whenever both rules were tested with the same n . Finally, Figure 146 shows mean run lengths for two lock-out attributes combined disjunctively. The growth rate is quadratic with a leading coefficient about half that of the single-attribute rule.

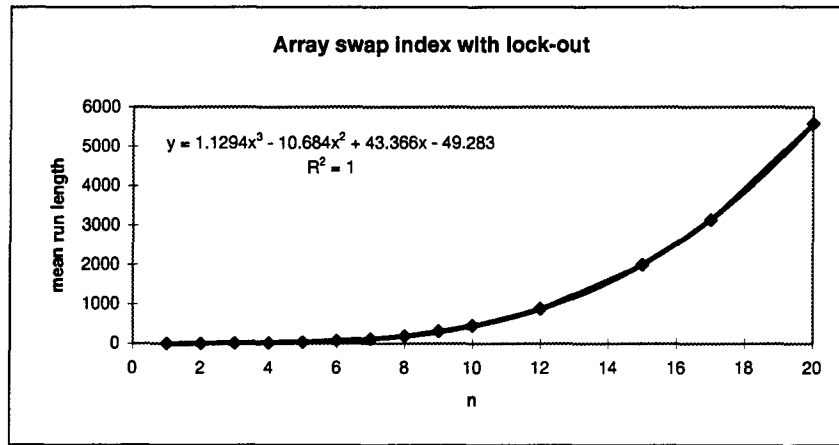


Figure 143. Mean Run Length for Array Swap Index with Lock-Out

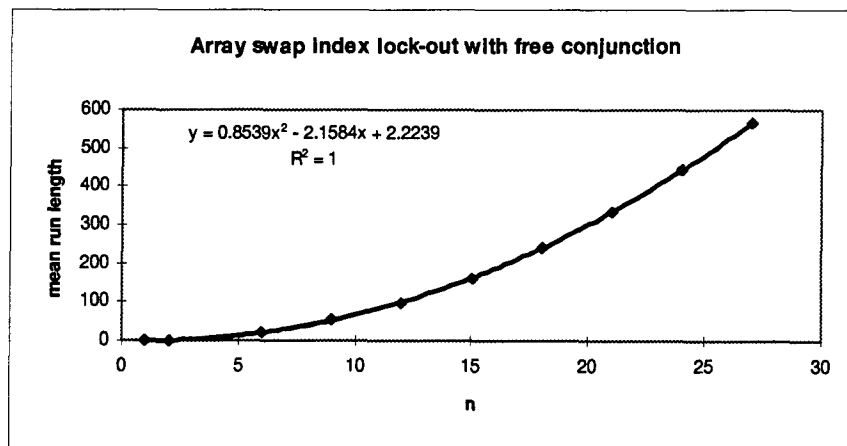


Figure 144. Mean Run Length for Array Swap Index with Free-Conjunctive Lock-Out

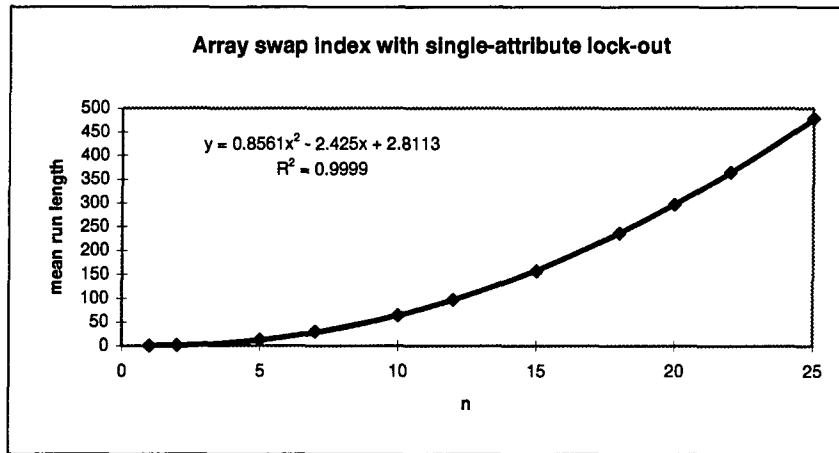


Figure 145. Mean Run Length for Array Swap Index with Single-Attribute Lock-Out

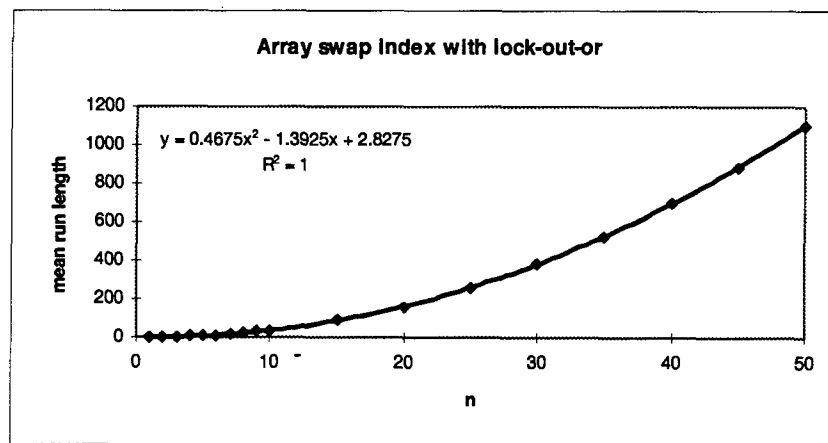


Figure 146. Mean Run Length for Array Swap Index with Disjunctive Lock-Out

These graphs show the differences between the various lock-out rules quite clearly. Lock-out rules permit a more intense search over an area of the solution space because of their low restrictiveness, yet still provide strong cycle prevention. Lock-in rules enforce somewhat more diversification by restricting the search more strongly. Depending on the problem being solved, in particular the relationship between the cost function and the neighborhood with respect to such things as the smoothness or spikiness of the topography and the number and distribution of local optima, either type of rule could be an effective choice for a tabu search. For the Kernighan-Lin heuristic, one or another lock-in rule is probably the best choice. Extremely long chains are computationally expensive and may not yield commensurately better solutions. Rules that generate long chains may be used, however, if chains are truncated by a length limit, as in the scheme presented in Section 7.2.1.

7.2.3 Experiments with Boolean Satisfaction. To explore the effectiveness of the Kernighan-Lin heuristic on a new problem, some limited experiments were done with boolean satisfaction. Hansen and Jaumard have performed more extensive experiments on boolean satisfaction, showing that Steepest Ascent Mildest Descent (i.e., tabu search) is both faster than simulated annealing and produces better quality solutions, and that both of these find better solutions than several constructive (i.e., non-search) heuristics (36). Selman et al. compare their algorithm with straight hill climbing (i.e., without neutral moves) and with the Davis-Putnam algorithm (a global search approach) (64). Inclusion of Kernighan-Lin is new.

Three programs were written in Refine. They were not derived formally in SPECWARE for a number of reasons: this work was done before program schemes were developed, code generation is a new feature we have not yet worked with much, the version of SPECWARE we are using is older and does not support code generation, and optimization of a program by algebraic means is not well understood and not supported by SPECWARE. The three programs implemented tabu search, the Kernighan-Lin heuristic and Selman's algorithm for boolean satisfiability using the

Algorithm	Tabu Search				Kernighan-Lin			Selman
Parameters	MaxMoves				Tie Breaking			MaxMoves
	20	40	60	80	Short	Random	Long	100
# solved/# trials	0/24	1/28	4/16	3/20	1/23	3/20	5/20	3/20
run time	1.5m	2.5m	3.5m	4.5m	4m	4m	4m	6m

Table 2. Comparing Three Local Search Algorithms for Boolean Satisfiability

map-set-var-to-val neighborhood. The *Val* sort is *Boolean*, so the lock-in and lock-out strategies for this problem are equivalent. The tabu search program used aspiration by global objective. One randomly-generated test formula was used. It has 50 variables and 215 clauses of 3 terms each. These values were chosen to make the test formula hard to solve, following (58).

Table 2 show the results obtained. Each entry in the table shows the number of times a solution that satisfied the test formula was found, the total number of trials run, and the average run time per trial (in minutes—Refine is a very high level language that is not intended for writing highly optimized and efficient code). For tabu search, the length of the tabu list is not shown: in the limited number of trials done, no differences attributable to tabu length were noticeable. The results shown combine trials with tabu lengths ranging from 2 to 25. The *MaxMoves* parameter limits the total number of moves made. This is not the *MaxNoChamp* parameter used in the program scheme. Optimal solutions were often found late in a search, so further increases in this parameter may increase the proportion of trials that find one. It may be more efficient, however, to keep the parameter relatively low and run more trials.

The Kernighan-Lin algorithm performed as well as the tabu search algorithm. A small detail in the implementation turned out to have a significant impact on the solution quality. At each step of the search a chain of 50 solutions is generated and a best solution selected as the new solution. If there is more than one best solution, the tie must be broken. If tie breaking always favors shorter chains by choosing the first best solution, performance is relatively poor. Breaking ties randomly is better, but the best results are achieved by always choosing the best solution that is last in the

chain. Given that all these solutions are equally good, it seems best to move as far as possible in each step so that future steps explore new regions of the solution space.

Selman's algorithm performed on a par with the others. Selman advocates setting *MaxMoves* to $10N$ or even $20N$, where N is the number of variables (64). We set it to only $2N$ to make its run time commensurate with the other algorithms. Note again that this parameter is not the one used in the program scheme for this algorithm, which limits the number of consecutive neutral moves, not the total number of moves.

Certain tentative conclusions can be drawn from this limited data. All three performed equally well, with some tuning in some cases. The Kernighan-Lin heuristic was gratifyingly competitive, given its simplicity and lack of tunable parameters. Chain lengths could be limited, but judging from the results of tabu search for shorter runs this would not be wise. Tabu search for this problem seems reasonably robust in terms of its parameters, but finding optimal settings is a difficult problem in general. The tabu approach in general gives more options, in terms of aspiration criteria and other techniques mentioned briefly at the beginning of Section 7.1 such as intermediate and long-term memory structures, target analysis and so on. Some of these, especially aspiration, could also be incorporated into Kernighan-Lin, if desired. Certainly the Kernighan-Lin heuristic, seen as a variant of tabu search, deserves further attention as a general technique for local search.

VIII. Conclusions

The goal of this investigation was to define a formal model of local search algorithms that supports their semi-automated synthesis. Wherever possible, methods were chosen or developed that would be broadly applicable to algorithm design in general and then illustrated by applying them to local search. The approach to algorithm design pioneered by KIDS was transplanted to the purely algebraic paradigm of SPECWARE largely intact, which is a tribute to the power and generality of that work and a major step in SPECWARE's planned evolution from a sophisticated and well-founded but largely passive diagramming tool to an active participant in the design process. Comments below about SPECWARE and Slang are meant as constructive suggestions for its continued development, not as criticism of its current state. Specific contributions of the research are discussed and evaluated next, followed by general comments, lessons learned, and recommendations for future work.

8.1 Contributions

The contributions that this research makes to the field of software engineering were outlined in Chapter I. The intervening chapters have fully developed methods for carrying out algorithm design in an algebraic setting and applying them to local search. The contributions may now be discussed and analyzed in more detail, as follows.

- *An algebraic theory of algorithm design.* Chapter IV describes a two-step process for designing algorithms that emphasizes precedented design and systematic reuse of software engineering knowledge. Problem classes represented as canonical forms are used to classify a problem, and program schemes parameterized on these canonical forms are instantiated with the specifics of a particular problem. The method exemplifies the "correct by construction" principle by applying verified components to formal problem specifications to produce both a program specification and an audit trail, in the form of an interpretation morphism, that the pro-

gram solves the problem. Knowledge is arranged hierarchically and applied incrementally. Principles of top-down design are supported and encouraged.

- *Methods for completing interpretation morphisms.* Chapter V identifies a family of methods for completing interpretation morphisms to or from a given interpretation and with respect to a given source morphism. The need for such methods was established in Chapter IV: both problem classification and program scheme instantiation involve completion of interpretation morphisms. Diagram refinement may also lead to instances of this problem. Most of the methods described are syntactic and so of limited use in design, though widely used for other purposes. Identity completion, for example, is the method of choice for instantiating a program scheme. The major contribution in this area is a new treatment of the connection mechanism, a semantic technique that is used by several of the design tactics of KIDS. An informal presentation provides the intuition behind the technique, while an algebraic formulation describes it formally. Polarity analysis rules are provided for features of the Slang language not previously considered. Guidelines for how connections are actually used to support design provide practical advice for applying the theory effectively. Examples in Chapters V and VI illustrate the process in full detail, while in Chapter VII the topic of extending a connection is discussed for the first time ever. Some questions about the proper application of polarity analysis of expressions involving higher-order operations remain open, but otherwise the theory developed completely characterizes the mechanism as described by Smith in (72) and as implemented in KIDS.
- *An informal characterization of local search.* In Chapter III a new framework is proposed whereby local search algorithms can be described and classified. The features identified apply to all local search algorithms and capture the essence of the technique. Each feature is analyzed in terms of its main variants in current practice, and a broad range of contemporary algorithms are described in terms of the framework. Similar characterizations have been proposed before, but with less detail and with less of a clear purpose in mind: the new

framework provides the basis for the formal treatment of local search in later chapters. It may also provide insight into local search that will lead to the invention of new variants and approaches.

- *A detailed analysis of neighborhood structures and their properties.* Neighborhood structure is the *sine qua non* of local search and Chapter VI provides an extensive and formal treatment of properties of neighborhoods. Many of these properties have been previously identified: reachability is widely discussed, while feasibility is usually assumed more or less implicitly. Exactness is also well known. Uniqueness seems to be new and is a technical property useful primarily in formal demonstrations. Formal treatments are less common but not unheard of; Lowry's is typical. The set of properties described here seems to be the most extensive one all in one place. Appendix A provides a sizeable collection of example neighborhoods, their properties, special features and relationships.
- *A design tactic for adapting neighborhood structures to problems.* Lowry proposed a design tactic for local search that left some doubt at the end as to which properties had been established and which had not. Building on the connection mechanism as described in Chapter V and the analysis of neighborhood properties, a new tactic is proposed that clearly identifies the properties that hold and so allows the user to trade them off or identify potential problems. The new tactic, like Lowry's, relies on a library of known neighborhood structures and reuses them by adapting them to new problems.

The new tactic is hampered by the strength of the connection condition generated by polarity analysis of the feasibility axiom. Several steps of the tactic are devoted to handling this problem. The result is largely specific to local search, but may serve as a model for other algorithm theories when similar difficulties arise.

- *Formalization of two basic local search approaches.* Basic local search, consisting of an optimization problem and a neighborhood structure, is sufficient to describe two program schemes

for local search. The first is hill climbing, the prototypical local search algorithm. A generic scheme for multi-trial strict hill climbing is presented and proved correct. This scheme is then specialized for the single-trial case and for steepest ascent, showing how program knowledge can be arranged hierarchically and applied incrementally. Methods for introducing new parameters in the program scheme are described, as well as the theory and practice of deferring certain design decision. The second program scheme is simulated annealing. Again a generic scheme that broadly characterizes the approach is provided, along with a specialization for run-based annealing. Conditions for proving that the algorithms terminate properly were lacking, however, preventing a formal proof of correctness.

- *Formalization of two advanced local search approaches.* Tabu search is a sophisticated approach to local search that is highly effective for a wide range of problems. Essential features of tabu search are identified in Chapter VII and formalized as extensions of local search. Tabu rules extend neighborhood structure with conditions for preventing search states from recurring; aspiration criteria provide supplementary conditions for overriding tabu status. Several tabu strategies are identified and applied to several example neighborhoods. Lock-in and lock-out, two very effective strategies, are described but not formalized. Considerations for discovering new tabu rules by modifying ones generated by the basic strategies are given, as well as strategies for managing multiple tabu lists. A number of specific aspiration criteria are described and formalized. A procedure for applying a tabu rule specialized to a library neighborhood by extending the connection built by the neighborhood tactic is proposed and illustrated by example. This procedure only works if the tactic yields a true connection but is otherwise general.

Chapter VII also describes the Kernighan-Lin heuristic, characterizing it as a specialization of tabu search. This is a new perspective on the technique and serves to generalize it to all problems to which tabu search is applicable. Tabu rules for two common neighborhoods are studied in detail to reveal characteristics relevant to their use in a Kernighan-Lin algorithm

and in more typical tabu algorithms. The effectiveness of the generalized Kernighan-Lin algorithm on the boolean satisfiability problem is demonstrated by a small empirical study, supporting the idea that this technique could be profitably applied to a wider range of problems than it has heretofore.

8.2 Discussion

The neighborhood design tactic and various other constructs are described as they would be applied by a person using SPECWARE as it currently exists. SPECWARE currently has no macro facility or scripting language that would directly support automating such recurring procedures. Indeed, invention of such a facility would be a substantial research effort if it is to provide the flexibility and interactivity needed. The only way to add new capabilities, then, is to modify SPECWARE itself, which is something best done by Kestrel. The theoretical work described above has been adequately demonstrated using the facilities already provided by SPECWARE to support the claim that automation of many aspects is possible and unlikely to encounter insurmountable obstacles, though doubtless much would be learned in the process.

Algorithm design relies heavily on stored knowledge systematically reused. This implies a knowledge acquisition problem: someone has to provide algorithm theories, program schemes, neighborhoods and other knowledge. All of this knowledge is stored declaratively and manipulated uniformly, making it very easy to expand the knowledge base. Algorithm theories and other problem classes encompass very broad classes of problems; one does not expect there to be a very large number of fundamental algorithmic approaches. Program schemes parameterized on algorithm theories describe specific methods for solving problems. The potential population of neighborhood theories is larger, but the tactic supports the development of general neighborhoods that can be adapted to many problems. Libraries also provide examples that future contributors to the knowledge base can use for guidance and inspiration.

In general the burden placed on those who populate the knowledge base should be much higher than the burden placed on those who use it, and this has been achieved. Proving that a program scheme is correct or that a neighborhood possesses certain properties may be difficult, but once done its cost is spread over its many uses. A designer must be familiar with the knowledge bases he or she will be using, to guide the system in making appropriate choices, for example, but the details of verifying the appropriateness of a choice can be supported with automation and in some cases need not involve the user at all. Since verified components are combined by methods that preserve correctness, consistency of the result with the specification is guaranteed. That is, the proposed design methodology does not eliminate the need for a knowledgeable software engineer, but it does provide a set of tools that can largely eliminate mistakes in the final product.

Many of the specs developed to describe problem classes and program schemes were not written in as general a form as possible. Chapter V briefly describes how in the specs *Problem* and *Program* the signatures for operations O and F could be changed to use the subsort $D \mid I$ rather than just D . This is sometimes necessary, if for example an invalid input would cause O or F to be undefined, say by requiring division by zero. A systematic effort to be as general as possible would cause subsorts to proliferate everywhere. The cost function in *WFSS*, for example, would most generally be defined only for solutions that are feasible with respect to a valid input, since non-feasible solutions may not have computable costs. The revised signature would be

$$\text{op Cost} : (D \mid I, R) \mid O \rightarrow \mathcal{R}$$

The domain sort, $(D \mid I, R) \mid O$, would probably be given an explicit name so it could be used by other operations conveniently, such as *N_info*. The signature of *N_is* would in turn use a subsort, characterized by *N_info*, for triples consisting of legal inputs, feasible solutions and valid transform values. The sort *Move* is an example of a subsort that could not be avoided; its characteristic function is essentially *N_is*. The sort *MSet*, a set of moves, could be made more precise by specifying not just a set of legal moves but a set of moves from a given feasible solution with respect

to a particular legal input. Program schemes would require specialized subsorts for the signatures of most operations. The advantage is that whoever writes definitions for these operations need not be concerned with their behavior on invalid inputs. Because subsorts have not been used extensively, the correctness proofs in Chapters VI and VII instead must always demonstrate that operations are defined for all inputs, valid or not, in addition to having the desired behavior for valid inputs.

The generality gained by systematic use of subsorts is at the cost of a profusion of specialized operations and sort axioms, and a constant need for relaxation, extra variables and the packing and unpacking of tuples in order to construct terms of the proper sort. Even trivial definitions become lengthy conditional expressions devoted mostly to binding variables. The comprehensibility of such definitions is severely compromised by formal manipulations that exist only to satisfy the type checker. The approach adopted in this research was to balance this complexity against the desire for generality. If the subsort mechanism were somehow made simpler and easier to use, permitting greater generality without a disproportionate increase in complexity, usability would be substantially increased.

Slang is not designed for convenient use as a programming language, yet to support the full software lifecycle it must at times be used as such, in particular during refinement, which includes algorithm design. Without compromising principles of abstraction and formal semantics, small changes could be made to Slang that would enhance its suitability to this role. One suggestion is a 'let' construct for introducing a variable and binding it to a value. This would presumably be treated as a kind of quantifier or defined in terms of existing ones. Conditional expressions (if-then-else) would make Slang more comfortable to users of more traditional languages, and is provided by other algebraic languages, such as Larch. Operation signatures could be made more self-documenting if names could optionally be given to the formal parameters. For example,

$$\text{op } \textit{TabuStep} : x : D, \textit{curr} : R, \textit{champ} : R, \textit{nochamp} : \textit{Nat}, \textit{tlen} : \textit{Nat}, \textit{TL} : \textit{TabuList}, \\ \textit{AM} : \textit{AspirationMap} \rightarrow R$$

seems superior to the signature used in Section 7.1.4.

At the spec level, a spec building operation that takes an existing spec and deletes certain components would have been quite handy. On several occasions, such as the clean-up rules proposed for use in identity completion of an interpretation morphism, the easiest way to construct a spec is to delete some elements from another. In general use, it is sometimes convenient to provide specs with a rich set of operations, such as for sets or integers. Adding operations one at a time produces a large number of specs and challenges one's ability to invent meaningful names. On the other hand, importing large specs with operations that are not needed is inconvenient and inefficient. The ability to provide rich theories and then tailor them by removing unneeded elements would assist users greatly.

An algebraic technique not supported by Slang is the designation of parts of a spec as being *hidden*. Sorts and operations declared hidden cannot be referred to outside of the spec that contains them. They are invisible to morphisms, for example, or when imported. Hidden parts provide another means for defining behavior abstractly. The domain theory for global search, for example, could make good use of hidden parts in defining *Split**. This operation is needed to characterize the desired behavior of the other elements, but the details of its definition are irrelevant. If *Split-K* and the spec *Nat* were hidden, several inconveniences in the *k*-queens example would have been avoided. Hidden parts also provide language-level support for information hiding, a principle not well supported by Slang. Specs may be structured as colimits, but once combined everything is visible to everything else. Interpretations provide a form of information hiding, in that the source spec describes "what" to do while the target spec and mediator tell "how" to do it in a way that is inaccessible to specs or diagrams incorporating the source, but the addition of hidden parts would provide another way to hide information.

Overall, the level of formal noise—constructs and manipulations that exist only to satisfy the mathematics and obscure the purpose that the mathematics is meant to serve—is very high in SPECWARE. The level of abstraction supported by algebraic methods is very high, but the amount

of detail involved in manipulating specs, morphisms and so on also remains high, and detail and complexity are what we are trying to control. When the detail is generated by the method rather than the problem being solved, it is noise and should be eliminated. The operations that SPECWARE provides are the proper foundation for software development but are too low-level to make for a practical tool. Evolution of the tool involves identifying and automating higher-level operations that are closer to the level of designers' thought processes.

One need felt during this research was for a greater understanding of parameterization. Chapter II illustrates how a colimit can be used to instantiate a parameter, but direct support for a formal notion of a parameterized spec as a first-class object with an instantiation operation would be better. More than this, all the constructs that SPECWARE provides—morphisms, colimits, interpretations, interpretation morphisms—must be defined for parameterized specs and provided with suitable operations that hide the details of how such things are composed, instantiated or otherwise manipulated. To the degree that program schemes can legitimately be viewed as parameterized on their respective algorithm theories, algorithm design would be enhanced by direct support for parameterized programming. Ehrig and Mahr have a notion of a *module* (which looks quite a bit like a parameterized interpretation) for which they define several high-level operations and show how to implement them in terms of colimits, composition of morphisms and so on (17). Their work serves as a model of defining high-level constructs and suppressing detail.

The value of this work in the short term is to provide a means for highly educated and motivated people to produce software where *high assurance* is required and high cost is not a major consideration. Algebraic algorithm design is not for the faint of heart, but neither is it an academic exercise: the technology is mature enough to begin moving it into practice. In the longer term, this work provides a foundation for continued work in algorithm design using formal methods, and a much-needed set of substantial examples that point out weaknesses in existing tools and methods.

8.3 Future Work

The research described in this dissertation has some loose ends that were discussed but not fully resolved when they arose. Several involve application of the connection mechanism. The first is the proper handling of higher-order constructs such as the built-in relaxation and quotient operations associated with subsorts and quotient sorts, respectively. The second is whether in a diagram refinement there is a clean formal mechanism that allows sharing in the mediator to be defined. The third is to figure out whether a tabu rule for a library neighborhood can still be used if the neighborhood has been modified via *FC* to adapt it to a problem. None of these seriously detract from the work that has been done, but they should be examined further in the near future.

Another outstanding task is to carry an example all the way through to code generation. As mentioned in Chapter VII, this capability only recently became available and the trauma of switching to a new version of SPECWARE was not judged worthwhile under the circumstances. Such an effort could be useful, however, in demonstrating that nothing significant has been overlooked and would allow us to compare synthesized algorithms with other implementations. To our knowledge, no one has yet attempted such a complex design refinement.

In the literature on local search, generic local search algorithms are often used to present and explain the essential elements of the technique. It might be useful to formalize such an algorithm, so that all other local search program schemes would be refinements of a single universal program scheme. Glover and Laguna give a pretty good rendition that might support such a formalization (27), and the program schemes presented in earlier chapters have enough in common to suggest that a universal scheme is possible. The domain theory would contain many significant design decisions that the various specific schemes would refine in their respective fashions. This would be an elegant theoretical result; the practical utility is likely to be limited. It might also prove difficult to prove termination in such a generic algorithm, a problem encountered in the program scheme for simulated annealing.

A more interesting set of problems concerns performing local search over neighborhoods that are infeasible. We mentioned in Chapter VI that feasibility is often sacrificed in order to guarantee reachability. This is often done by relaxing the problem and modifying the cost function. Is it possible to use the feasibility constraint, FC , to suggest such a reformulation? For some problems and neighborhoods one can prove that all local optima are feasible even though the neighborhood as a whole includes infeasible solutions. In graph theory, finding a maximal independent set is a problem of this type. Here the objective is to find the largest set of nodes such that there are no edges between any nodes of the set. A common neighborhood for this problem is to add a node to the current solution and then remove all nodes adjacent to it. This neighborhood is reachable over the set of maximal independent sets, but it is not feasible because not all subsets generated this way are maximal. All locally optimal solutions are feasible, however, so any algorithm that guarantees to find a local optimum will return a feasible solution without having to reformulate the problem or modify the neighborhood. It might be useful to check for this phenomenon in the neighborhood tactic.

Lin and Kernighan describe a heuristic for the symmetric traveling salesman problem that uses an infeasible neighborhood but always produces feasible solutions (49). It is similar to the Kernighan-Lin algorithm in that it chains together moves looking for improved solutions, but the chains may contain intermediate solutions that are infeasible. Each solution in the chain is always at most one move away from some feasible solution, and at each link the algorithm decides whether it is better to *close* the chain by moving to that feasible solution or to make some other move to another infeasible one. Like Kernighan-Lin, the Lin-Kernighan algorithm produces some of the best results known but has not been applied to other problems. Identifying the problem features required for this approach would generalize it significantly. Finally, it has been claimed that tabu search can be adapted to infeasible neighborhoods and even use them strategically (28), but we are aware of no examples.

Tabu search presents many promising avenues for future research in algebraic algorithm design. Reverse elimination and cancellation sequences are two methods that evaluate chains of moves to see if they regenerate a solution visited in the past, and makes such chains tabu by forbidding their last moves (24). These techniques are efficiently implementable yet the only moves classified tabu are those that actually revisit a solution. Formalizing these techniques would require identifying the special neighborhood characteristics that they depend on.

Other techniques commonly grouped under the "tabu" rubric could be formalized, such as the use of intermediate and long-term memory structures for various purposes. These typically make use of frequency-based information to devise weighting functions that influence the selection of moves or initial solutions, for example by modifying the cost function (67).

An important research area that was encountered many times while working on local search is formal support for design optimization. Many times in writing problem specifications, program schemes, neighborhoods, tabu rules and so on there have been choices between representing something abstractly or taking advantage of some circumstance to do it more efficiently. In most cases the abstract representation was chosen, deferring optimization to later design steps. KIDS provides a nice environment for program optimization, with powerful operations for partial evaluation, finite differencing and context-dependent simplification. These aspects of KIDS also need to be carried over to SPECWARE so that generated code is efficient as well as correct. This is a significant research area in its own right, independent of local search or even algorithm design. Optimizations in KIDS transform code destructively, replacing the old version with the new. This is probably the wrong paradigm for an algebraic system. We want to modify the internal structure of a spec to produce a new spec that is isomorphic to it, but it seems better to produce a new spec rather than modify a spec. The isomorphism is actually between the theories generated by the specs: it may require interpretations to demonstrate the isomorphism. Thus optimization operations might be used as spec building operations to generate both a new spec and a refinement to or from the old one, just

as import and translate each generate both specs and morphisms. Unskolemization might also be implemented this way, adding a definition and deleting an axiom (or changing it to a theorem) in one step.

Finally, Appendix C contains a discussion of a theory of neighborhood structure proposed by Lowry (51) that includes suggestions for future research in this area.

Appendix A. Library of Neighborhood Structures

This appendix provides a library of neighborhood structures over common data types. With each is a brief description of the solution space over which the neighborhood ranges, properties of the neighborhood, representative problems to which the neighborhood has been applied, and relationships to other neighborhoods. Ways in which the tabu strategies described in Chapter VII can be applied are also discussed.

As illustrated in Chapter VI, the formal representation of a neighborhood structure is as an interpretation morphism from a problem specification to a neighborhood specification. This is the form assumed by the neighborhood tactic, for example. This representation is inconvenient for the immediate purpose, however, which is primarily to provide a range of neighborhoods to the reader in order to make concrete the properties that neighborhoods have and build up intuition, and only secondarily to populate an actual knowledge base. A full neighborhood presentation requires up to four specs to be defined, defining two interpretations and a morphism between them. The target specs of the interpretations often require complex diagrams to build up descriptions of data structures. Each neighborhood would thus require several pages to present and its essential structure would be obscured by the algebraic detail.

To simplify the presentation of the neighborhood, we will instead adopt a KIDS-like notation for interpretations and will present only the neighborhood specification, since it contains the specification of the feasibility problem within it. Further, we will assume the existence of domain theories containing basic facts about sets, arrays, finite maps, arithmetic, etc. Parameter sorts, such as the sort of the elements in a set, will be indicated by Greek letters or other symbols given after the sort name, such as *Set*(α). Interpretations will be presented as mappings from the sort and operation symbols of *Neighborhood* (which are *D*, *R*, *I*, *O*, *C*, *N_is*, and *N_info*) using unnamed expressions from this domain theory. The symbols *N* and *N** have definitions in the spec and these will not be repeated. The translation of this form to Slang is straightforward. Additional simplifications are

sometimes made to present the neighborhoods clearly at the expense of strict correctness; these are pointed out in the text.

A.1 *k*-Subset-1-Exchange

k-subset-1-exchange is a perfect, symmetric neighborhood for searching over fixed-size subsets of a set. This neighborhood was presented formally in Chapter VI. It is used for many graph problems, such as graph partitioning and other problems involving finding sets of nodes or edges with certain properties. The spanning trees of a graph, for example, can be searched over by restricting *N_info* with a feasibility constraint that insures that the edge dropped from the current solution is on the cycle created by the edge added. The traveling salesman problem can be viewed as finding a subset of edges that constitute a tour. *FC* for this problem is *false*: there is no way to drop and add one pair of edges from a tour and still have a tour. Chains of two or more swaps, however, define a perfect, symmetric neighborhood for the traveling salesman. The simplex algorithm for linear programs also uses this neighborhood, searching over sets of basic variables that correspond to feasible solutions.

$$\begin{aligned}
 D &\mapsto Nat, Set(\alpha) \\
 I &\mapsto \lambda(k, S) (k \leq size(S)) \\
 R &\mapsto Set(\alpha) \\
 O &\mapsto \lambda(\langle k, S \rangle, A) (k = size(A) \wedge A \subseteq S) \\
 C &\mapsto \alpha, \alpha \\
 N\_info &\mapsto \lambda(\langle k, S \rangle, A, \langle i, j \rangle) (i \in S \setminus A \wedge j \in A) \\
 N\_is &\mapsto \lambda(\langle k, S \rangle, A, \langle i, j \rangle, new\_A) \\
 &\quad (new\_A = (A \cup \{i\}) \setminus \{j\})
 \end{aligned}$$

Figure 147. *k*-Subset-1-Exchange

A subset can also be represented by its characteristic function, a boolean map that assigns *true* to elements in the subset and *false* to the rest. *k*-subset-1-exchange is then equivalent to map-swap-domain over the characteristic function, since the size of the subset is fixed.

Tabu rules for this neighborhood were discussed in Sections 7.1.1 and sec:ts-ks1e. The lock-in, lock-out and inverse move strategies all yield distinct rules, and further variations are found

by analyzing these rules. The behavior of many of these rules was examined analytically and empirically in the context of the Kernighan-Lin heuristic in Section 7.2.2.1.

A.2 *k*-Subset-2-Exchange

k-subset-2-exchange is another perfect, symmetric neighborhood for fixed-size subsets of a set, under slightly restricted conditions. If *k* equals 0 or *size*(*S*), then there is only one solution and no moves can be made. If *k* equals 1 or *size*(*S*) - 1, however, there are *size*(*S*) solutions but still no moves defined by this neighborhood because there is no way to swap two elements simultaneously. For less extremal *k*, the neighborhood is reachable unless *size*(*S*) = 4. If *S* = {*a*, *b*, *c*, *d*} and *k* = 2, for example, the solution {*a*, *b*} has only one neighbor, namely {*c*, *d*}, whose only neighbor is again {*a*, *b*}. The other four solutions are not reachable from these two, but constitute two more distinct orbits of two solutions each.

$$\begin{aligned}
 D &\mapsto Nat, Set(\alpha) \\
 I &\mapsto \lambda(k, S) (k \neq 1 \wedge k \neq size(S) - 1 \wedge size(S) \neq 4) \\
 R &\mapsto Set(\alpha) \\
 O &\mapsto \lambda(\langle k, S \rangle, A) (k = size(A) \wedge A \subseteq S) \\
 C &\mapsto \alpha, \alpha, \alpha, \alpha \\
 N\_info &\mapsto \lambda(\langle k, S \rangle, A, \langle i, j, m, n \rangle) \\
 &\quad (i \in S \setminus A \wedge j \in S \setminus A \wedge i \neq j \wedge m \in A \wedge n \in A \wedge m \neq n) \\
 N\_is &\mapsto \lambda(\langle k, S \rangle, A, \langle i, j, m, n \rangle, new\_A) \\
 &\quad (new\_A = (A \cup \{i, j\}) \setminus \{m, n\})
 \end{aligned}$$

Figure 148. *k*-Subset-2-Exchange

This neighborhood is bigger than *k*-subset-1-exchange, in that each solution has more neighbors, and a move makes a bigger change to a solution. It can be used for any problem to which *k*-subset-1-exchange is appropriate. This neighborhood can also be used for the traveling salesman problem with a feasibility constraint that determines the edges to be added from the edges to be dropped so that a tour is produced. A 2-exchange is equivalent to a 2-chain of 1-exchanges. The generalization to 3-exchanges and beyond is obvious. Lin showed that for the traveling salesman, 3-exchange produces much higher quality tours than 2-exchange (48), while the Lin-Kernighan algorithm uses variable-length chains to do even better (49).

Meaningful tabu rules for this neighborhood parallel those for k -subset-1-exchange. Lock-in would require both added elements to remain in future solutions and both dropped elements to remain out, while lock-out would permit these elements to move in and out as long as at least one added element remained in or one dropped element remained out. A wide variety of intermediate rules is possible, such as focusing only on added elements, or only on dropped ones, or using different tenures for added and dropped elements. Inverse moves exist, but do not prevent solutions from recurring.

A.3 Free-Subset

The free-subset neighborhood is a perfect, symmetric neighborhood for searching over all subsets of a set. The transform sort is a single set element. In effect the position of element i is “toggled” between being in the current solution and being out. If the size of the set is n , then each subset has n neighbors.

$$\begin{aligned}
 D &\mapsto \text{Set}(\alpha) \\
 I &\mapsto \lambda(S) \text{ true} \\
 R &\mapsto \text{Set}(\alpha) \\
 O &\mapsto \lambda(S, A) (A \subseteq S) \\
 C &\mapsto \alpha \\
 N\_info &\mapsto \lambda(S, A, i) (i \in S) \\
 N\_is &\mapsto \lambda(S, A, i, new\_A) \\
 &\quad (new\_A = \text{if } i \in A \text{ then } A \setminus \{i\} \text{ else } A \cup \{i\})
 \end{aligned}$$

Figure 149. Free-Subset

The k -subset-1-exchange neighborhood can be viewed as a chain of two free moves, constrained so that one is an add and one is a drop. If the subset is represented by its characteristic function, this neighborhood is equivalent to map-set-var-to-val. Knapsack and other 0-1 integer programs can be approached with this neighborhood. Problems where a subset of a particular size is desired can be relaxed to use it by adding a penalty term to the cost function that measures the deviation of the current solution from the desired size. Graph partitioning with the Kernighan-Lin heuristic has been approached this way (18), with slightly lower-quality solutions produced in less time than the usual implementation (15).

It turns out that the lock-in, lock-out and inverse move tabu strategies all yield the same rule: once an element is moved, it cannot be moved again. This simple rule has no obvious relaxations, though it may be useful to distinguish which moves are adds and which are drops. The “influence” of a move, in terms of the restrictiveness of its tabu status, depends on the size of

the current solution, which varies. A clever list management strategy might be able to exploit this characteristic, perhaps by varying tabu list lengths systematically.

A.4 Independent-Set

Independent-set is a neighborhood for searching over the independent sets of an undirected graph $G = (V, E)$, represented here by an adjacency matrix, A . Legal matrices are described using properties like those defined for cost matrices in Section 6.2.4. An *independent set* of G is a subset of the nodes of G such that no two nodes in the subset have an edge between them. The neighborhood is written to allow loops (edges where both endpoints are the same node), but nodes with loops will of course never appear in a solution. A transform value is a single node that is not in the current solution and has no loops. This node is added, and all nodes in the current solution that are adjacent to it are removed. N_is below is written using a set former in Refine syntax. This neighborhood is not symmetric, since there is no way to add in a single move more than one node just dropped. The size of the neighborhood varies: it is the number of nodes not in the current solution.

$$\begin{aligned}
 D &\mapsto \text{Map}((\text{Node}, \text{Node}), \text{Boolean}) \\
 I &\mapsto \lambda(A) (\text{square}(A) \wedge \text{symmetric}(A)) \\
 R &\mapsto \text{Set}(\text{Node}) \\
 O &\mapsto \lambda(A, S) S \subseteq \text{Vertices}(A) \\
 &\quad \wedge \forall(x, y) (x \in S \wedge y \in S \Rightarrow \neg A(x, y)) \\
 &\quad \wedge \forall(x) (x \in \text{Vertices}(A) \setminus S \\
 &\quad \quad \Rightarrow \exists(y) (y \in S \wedge (A(x, y) \vee A(y, y)))) \\
 C &\mapsto \text{node} \\
 N\_info &\mapsto \lambda(A, S, i) (i \in \text{Vertices}(A) \setminus S \wedge \neg A(i, i)) \\
 N\_is &\mapsto \lambda(A, S, i, \text{new\_}S) \\
 &\quad (\text{new\_}S = \{x \mid (x) x \in S \cup \{i\} \wedge \neg A(x, i)\})
 \end{aligned}$$

Figure 150. Independent-Set

The output condition O describes a *maximal* independent set, which is an independent set that is not a proper subset of any other independent set. This neighborhood is reachable for this solution space but is not feasible since adding a node to the current solution can cause several nodes to be deleted such that the result is not maximal. Maximality is usually incorporated into

the cost function, as for the weighted independent set problem described below, but it is nonetheless important that all maximal solutions are reachable.

With respect to the set of all independent sets of a graph, this neighborhood is feasible but not reachable. To search over all independent sets one needs the ability to remove a node from the current solution. It may be possible to force a node out by adding an adjacent one, but in general there is no guarantee that any sequence of moves exists from a solution to a proper subset of it. One could expand the neighborhood definition to allow this kind of move, but for many applications this is not necessary. The weighted independent set problem, for example, assigns a weight to each node and defines the cost of a solution as the sum of the weights of the nodes in it (42). As long as all weights are positive, all global and local optima will be maximal independent sets, so even though some solutions are not reachable, all potentially optimal ones are, and any program scheme guaranteed to return a locally optimal solution will return a maximal one.

A special case of the weighted independent set problem is finding the largest maximal independent subset (i.e., all nodes are assigned a weight of one). Solutions to this problem can in turn be applied to finding cliques (maximal complete subgraphs) in the complement of G (20, 21). This problem is also related to graph coloring, which seeks to partition the nodes of a graph into the smallest number of independent subsets (9).

Moves for this neighborhood can be viewed as variable-length chains of free-subset moves, with additional conditions added. Tabu rules parallel those for the other subset neighborhoods above: lock-in requires the entire effect of a move to be maintained, while lock-out prevents only the reoccurrence of the entire state before the move. Lock-in is likely to be very restrictive since each move affects many elements, and for the same reason lock-out seems likely to be very unrestrictive. A variety of intermediate rules is possible, and added and dropped elements can again be treated separately, with different tenures. Nodes of high degree in general tend to have a greater effect on the current solution than nodes of low degree, and the effect of a given node can vary greatly depending

on which other nodes are in the current solution. Since the neighborhood is not symmetric, there is no inverse move strategy.

A.5 2-Partition-1-Exchange

2-partition-1-exchange is a perfect, symmetric neighborhood for searching over partitions of a set into two subsets of specified sizes. This is completely equivalent to k -subset-1-exchange, which can be viewed as returning one element of such a partition and implicitly identifying the other by exclusion. The elements of a partition P are shown being accessed using Refine's dot notation, as $P.1$ and $P.2$. The corresponding Slang expressions are **(project 1)(P)** and **(project 2)(P)**, respectively. If the two specified sizes are k and l , then the size of the neighborhood is $k \cdot l$. Tabu rules for 2-partition-1-exchange are the same as for k -subset-1-exchange.

$$\begin{aligned}
D &\mapsto \text{Set}(\alpha), \text{Nat}, \text{Nat} \\
I &\mapsto \lambda(S, k, l) (k + l = \text{size}(S)) \\
R &\mapsto \text{Set}(\alpha), \text{Set}(\alpha) \\
O &\mapsto \lambda(\langle S, k, l \rangle, P) (P.1 \subseteq S \wedge P.2 \subseteq S \\
&\quad \wedge \text{size}(P.1) = k \wedge \text{size}(P.2) = l \\
&\quad \wedge \text{disjoint}(P.1, P.2)) \\
C &\mapsto \alpha, \alpha \\
N\_info &\mapsto \lambda(\langle S, k, l \rangle, P, \langle i, j \rangle) (i \in P.1 \wedge j \in P.2) \\
N\_is &\mapsto \lambda(\langle S, k, l \rangle, P, \langle i, j \rangle, \text{new\_}P) \\
&\quad (\text{new\_}P = (\langle P.1 \cup \{j\} \rangle \setminus \{i\}, \langle P.2 \cup \{i\} \rangle \setminus \{j\}))
\end{aligned}$$

Figure 151. 2-Partition-1-Exchange

One could define similar neighborhoods for partitions with three elements and their sizes, or some other specific number of elements, or for arbitrary partitions, where the input is a bag of sizes. When the partition elements are of no particular size, a “free” neighborhood could be defined, in direct correspondence with the free neighborhood for subsets defined above. Graph coloring can be defined as returning a partition of minimum size, where each element of the partition is an independent set of vertices (see Section A.4) and a free neighborhood used that moves a node from one partition element to another, possibly deleting or creating an element in the process (9). Another generalization is to define 2-exchange neighborhoods and beyond.

A.6 Map-Swap-Domain

Map-swap-domain is a perfect, symmetric neighborhood for searching over maps with a specified domain and range-bag. Input is a set, which is the domain of the map, and a bag, which is the range of the map with duplicates allowed. The set and the bag must have the same size. Output is a map. Transform values are pairs of domain elements. A map is transformed by exchanging the range elements assigned to the two domain elements. The map must assign distinct range elements to the two domain elements for the transform to be legal. Two versions of *N_{is}* are given. The first uses an operation defined immediately following that describes how to swap two elements (it is defined even for maps that are not defined at *i* or at *j*). The second is a predicate describing more abstractly the effect of the transformation: that the old and new maps are identical everywhere except for the swap. The second form is more suggestive of possible tabu rules. The transform values $\langle i, j \rangle$ and $\langle j, i \rangle$ have the same effect on a solution, so for uniqueness a quotient sort could be used to eliminate the redundancy. If the size of the domain set is n , then each map has $\frac{1}{2}n(n-1)$ neighbors.

Problems involving bijections from one set to another often use this neighborhood because it preserves the bijectivity. Examples include the k -queens problem, used to illustrate connections in Chapter V, and other constraint satisfaction problems that use bijective maps (82). These include assignment problems, such as the quadratic assignment problem (67).

This neighborhood is closely related to array-swap-index, described in Section 7.1.2.3 and below in Section A.9. Array-swap-index treats an array as a map, using *Nat* as the domain sort. Tabu rules for map-swap-domain are virtually identical to those for array-swap-index.

```

D  ↦  Set( $\alpha$ ), Bag( $\beta$ )
I  ↦   $\lambda(S, R) (size(S) = size(R))$ 
R  ↦  Map( $\alpha, \beta$ )
O  ↦   $\lambda(\langle S, R \rangle, M) (domain(M) = S \wedge range-bag(M) = R)$ 
C  ↦   $\alpha, \alpha$ 
N_info ↦  $\lambda(\langle S, R \rangle, M, \langle i, j \rangle) (i \in S \wedge j \in S \wedge M(i) \neq M(j))$ 
N_is  ↦  $\lambda(\langle S, R \rangle, M, \langle i, j \rangle, new\_M)$ 
       $(new\_M = swap-domain(M, i, j))$ 
      OR
       $(new\_M(i) = M(j) \wedge new\_M(j) = M(i)$ 
       $\wedge \forall(k) (k \in S \wedge k \neq i \wedge k \neq j \Rightarrow new\_M(k) = M(k)))$ 

% Exchange values mapped to domain elements  $i$  and  $j$ 
op swap-domain : Map,  $\alpha, \alpha \rightarrow$  Map
definition of swap-domain is
  axiom  $\forall(i : \alpha, j : \alpha) (swap-domain(empty-map, i, j) = empty-map)$ 
  axiom  $\forall(M : Map, b : \beta, i : \alpha, j : \alpha)$ 
     $(swap-domain(map-shadow(M, i, b), i, j)$ 
     $= map-shadow(swap-domain(M, i, j), j, b))$ 
  axiom  $\forall(M : Map, b : \beta, i : \alpha, j : \alpha)$ 
     $(swap-domain(map-shadow(M, j, b), i, j)$ 
     $= map-shadow(swap-domain(M, i, j), i, b))$ 
  axiom  $\forall(M : Map, a : \alpha, b : \beta, i : \alpha, j : \alpha)$ 
     $(a \neq i \wedge a \neq j$ 
     $\Rightarrow swap-domain(map-shadow(M, a, b), i, j)$ 
     $= map-shadow(swap-domain(M, i, j), a, b))$ 
end-definition

```

Figure 152. Map-Swap-Domain

A.7 Map-Swap-Range

Map-swap-range is a perfect, symmetric neighborhood for searching over maps with a specified domain-partition and into a specified range. This neighborhood is dual to the map-swap-domain neighborhood described in the previous section. A map implicitly defines an equivalence relation over its domain when two domain elements are considered equivalent whenever the map assigns them both the same range value. This equivalence relation partitions the domain element into disjoint classes. A map is transformed by identifying two range elements, i and j , changing the map at all domain elements mapped to i to map them to j , and changing the map at all domain elements mapped to j to map them to i . This transformation preserves the domain partition. A transform can change the range of the map, in the elements it contains or their multiplicity, but if range elements are always chosen from a set Q , the range will always be a subset of Q . At least one of i or j must already be in the range of a map for the transform to be legal (if neither is in the range, the transform has no effect). The transform values $\langle i, j \rangle$ and $\langle j, i \rangle$ have the same effect on a solution, so for uniqueness a quotient sort could be used to eliminate the redundancy. If the size of the range set is n , then each map has $\frac{1}{2}n(n - 1)$ neighbors, minus one for each pair where neither is in the range of the map.

Note that for a bijective map, swapping the range is the same as swapping the domain since the domain partition consists of singleton sets. In general, if the generalized inverse of a map from α to β is defined to return a map from β to $set(\alpha)$, mapping each range element to the set of domain elements that are mapped to it in the original map, then the following relationship holds between map-swap-domain and map-swap-range:

$$inverse(swap-range(A, i, j)) = swap-domain(inverse(A), i, j)$$

Applications of map-swap-range appear to be rare. It is provided in the library for completeness, as a complement to map-swap-domain. Possible tabu rules include those for map-swap-domain applied to the inverse map.

$$\begin{aligned}
D &\mapsto \text{Set}(\text{Set}(\alpha)), \text{Set}(\beta) \\
I &\mapsto \lambda(P, Q) (\text{size}(P) \leq \text{size}(Q) \\
&\quad \wedge \forall(A, B) (A \in P \wedge B \in P \Rightarrow \text{disjoint}(A, B))) \\
R &\mapsto \text{Map}(\alpha, \beta) \\
O &\mapsto \lambda(\langle P, Q \rangle, M) (\text{range}(M) \subseteq Q \\
&\quad \wedge \text{domain-partition}(M) = P) \\
C &\mapsto \beta, \beta \\
N\_info &\mapsto \lambda(\langle P, Q \rangle, M, \langle i, j \rangle) (i \in Q \wedge j \in Q \wedge i \neq j \\
&\quad \wedge (i \in \text{range}(M) \vee j \in \text{range}(M))) \\
N\_is &\mapsto \lambda(\langle P, Q \rangle, M, \langle i, j \rangle, \text{new\_}M) \\
&\quad (\text{new\_}M = \text{swap-range}(M, i, j))
\end{aligned}$$

% Exchange the sets of domain elements mapped to range elements i and j

op *swap-range* : $\text{Map}, \beta, \beta \rightarrow \text{Map}$

definition of *swap-range* **is**

axiom $\forall(i : \beta, j : \beta) (\text{swap-range}(\text{empty-map}, i, j) = \text{empty-map})$

axiom $\forall(M : \text{Map}, a : \alpha, i : \beta, j : \beta)$
 $(\text{swap-range}(\text{map-shadow}(M, a, i), i, j)$
 $= \text{map-shadow}(\text{swap-range}(M, i, j), a, j))$

axiom $\forall(M : \text{Map}, a : \alpha, i : \beta, j : \beta)$
 $(\text{swap-range}(\text{map-shadow}(M, a, j), i, j)$
 $= \text{map-shadow}(\text{swap-range}(M, i, j), a, i))$

axiom $\forall(M : \text{Map}, a : \alpha, b : \beta, i : \beta, j : \beta)$
 $(b \neq i \wedge b \neq j$
 $\Rightarrow \text{swap-range}(\text{map-shadow}(M, a, b), i, j)$
 $= \text{map-shadow}(\text{swap-range}(M, i, j), a, b))$

end-definition

Figure 153. Map-Swap-Range

A.8 Map-Set-Var-to-Val

Map-set-var-to-val is a perfect, symmetric neighborhood for searching over maps with a specified domain and into a specified range. The problem specification *EnumerateMaps*, defined in Section 5.2.5, describes the solution space over which this neighborhood ranges. A transform value is a pair $\langle a, b \rangle$ where a is in the domain set, b is in the range set and $M(a) \neq b$. *Map-shadow* plays the role of *N_is*. A second form for *N_is* is shown that highlights the differences between neighbors. A map-swap-domain move can be viewed as a special case of a 2-chain of map-set-var-to-val moves. If the size of the domain set is n and the size of the range set is m , then a map has $n(m - 1)$ neighbors. Map-set-var-to-val is a common neighborhood for constraint satisfaction problems that do not involve bijective maps, such as boolean satisfiability. Tabu rules for this neighborhood were discussed in Section 7.1.2.4.

$$\begin{aligned}
 D &\mapsto \text{Set}(\alpha), \text{Set}(\beta) \\
 I &\mapsto \lambda(Q, R) \text{ true} \\
 R &\mapsto \text{Map}(\alpha, \beta) \\
 O &\mapsto \lambda(\langle Q, R \rangle, M) (\text{domain}(M) = Q \wedge \text{range}(M) \subseteq R) \\
 C &\mapsto \alpha, \beta \\
 N\_info &\mapsto \lambda(\langle Q, R \rangle, M, \langle a, b \rangle) (a \in Q \wedge b \in R \wedge M(a) \neq b) \\
 N\_is &\mapsto \lambda(\langle Q, R \rangle, M, \langle a, b \rangle, \text{new\_M}) \\
 &\quad (\text{new\_M} = \text{map-shadow}(M, a, b)) \\
 &\quad \text{OR} \\
 &\quad (\text{new\_M}(a) = b \\
 &\quad \wedge \forall(w) (w \in Q \wedge w \neq a \Rightarrow \text{new\_M}(w) = M(w)))
 \end{aligned}$$

Figure 154. Map-Set-Var-to-Val

A.9 Array-Swap-Index

Array-swap-index is a perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements. This neighborhood, including a large collection of tabu rules, is described in detail in Section 7.1.2.3. As noted above, array-swap-index treats an array as a map and so is directly comparable to map-swap-domain. The sort C is shown with the quotient omitted, and the operation *swap-index* is shown with bounds checking omitted. If the size of the bag is n , then each array has $\frac{1}{2}n(n-1)$ neighbors.

$$\begin{aligned}
 D &\mapsto \text{Bag}(\alpha) \\
 I &\mapsto \lambda(R) \text{ true} \\
 R &\mapsto \text{Array}(\alpha) \\
 O &\mapsto \lambda(R, A) (\text{range-bag}(A) = R) \\
 C &\mapsto \text{Nat}, \text{Nat} \\
 N\_info &\mapsto \lambda(R, A, \langle i, j \rangle) \\
 &\quad (i < \text{size}(R) \wedge j < \text{size}(R) \wedge A(i) \neq A(j)) \\
 N\_is &\mapsto \lambda(R, A, \langle i, j \rangle, \text{new\_}A) (\text{new\_}A = \text{swap-index}(A, i, j)) \\
 &\quad \text{OR} \\
 &\quad (\text{new\_}A(i) = A(j) \wedge \text{new\_}A(j) = A(i) \\
 &\quad \wedge \forall (k : \text{Nat}) (k < \text{size}(R) \wedge k \neq i \wedge k \neq j \Rightarrow \text{new\_}A(k) = A(k)))
 \end{aligned}$$

```

% Exchange positions of elements at  $i$  and  $j$ 
op swap-index : Array, Nat, Nat  $\rightarrow$  Array
definition of swap-index is
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Nat})$ 
    (swap-index( $A, i, j$ ) = update-array(update-array( $A, i, A(j)$ ),  $j, A(i)$ ))
end-definition

```

Figure 155. Array-Swap-Index

This neighborhood is widely used, for example in job-shop problems (19) and other sequencing or ordering problems. An array can be used to represent a tour of a graph, and this neighborhood has been applied to the traveling salesman problem, but with poor results (19). Lowry has shown how sorting can be defined as an optimization problem over arrays and, using array-swap-index, derives the bubble sort algorithm (51).

A.10 Array-Swap-Adjacent-Index

Array-swap-adjacent-index is another perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements. It is smaller than array-swap-index; in particular, the $n - 1$ neighbors of an array according to the former are a subset of the neighbors according to the latter. This neighborhood could be defined simply by adding a condition to N_info as defined for the array-swap-index neighborhood, but instead it is a new neighborhood specialized to take advantage of the added structure (e.g., by simplifying C).

$$\begin{aligned}
 D &\mapsto Bag(\alpha) \\
 I &\mapsto \lambda(R) true \\
 R &\mapsto Array(\alpha) \\
 O &\mapsto \lambda(R, A) (range\_bag(A) = R) \\
 C &\mapsto Nat \\
 N\_info &\mapsto \lambda(R, A, i) (i < size(R) - 1) \\
 N\_is &\mapsto \lambda(R, A, i, new\_A) \\
 &\quad (new\_A = swap\_index(A, i, i + 1))
 \end{aligned}$$

Figure 156. Array-Swap-Adjacent-Index

Note that the concept of adjacency relies on the ordering of the natural numbers that are used to index the array. The array is still largely being treated as a map, with the exception that map domains in general do not have such an order. Tabu rules for array-swap-index can be optimized somewhat to take advantage of the additional structure, but otherwise are the same. The inverse of move i , for example, is the move i , while the lock-in rule would forbid moves $i - 1$, i , and $i + 1$, assuming all of these are valid moves, in order to prevent the swapped elements from moving again.

Applications are the same as for array-swap-index: job-shop scheduling and other sequencing and ordering problems. Search proceeds more quickly for this neighborhood, but for some problems the solution quality suffers.

A.11 Array-Insert-Element

Array-insert-element is another perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements. A transform variable is a pair of indices, $\langle i, j \rangle$, in the bounds of the array and distinct. A transform variable is applied to an array by moving the element at index j to index i and moving the elements at positions i through $j - 1$ up (if $i < j$) or elements at $j + 1$ through i down (if $j < i$) one index position. This can be implemented using operations *insert-at* and *delete*, which alter the size of an array.

$$\begin{aligned}
 D &\mapsto Bag(\alpha) \\
 I &\mapsto \lambda(R) true \\
 R &\mapsto Array(\alpha) \\
 O &\mapsto \lambda(R, A) (range\_bag(A) = R) \\
 C &\mapsto Nat, Nat \\
 N\_info &\mapsto \lambda(R, A, \langle i, j \rangle) \\
 &\quad (i < size(R) \wedge j < size(R) \wedge i \neq j) \\
 N\_is &\mapsto \lambda(R, A, \langle i, j \rangle, new\_A) \\
 &\quad (new\_A = insert\_at(delete(A, j), i, A(j)))
 \end{aligned}$$

Figure 157. Array-Insert-Element

Even though index positions are used to characterize the neighborhood, the array is being viewed more as a sequence rather than a map. A move causes many elements to change absolute positions, but only a few have their immediate neighbors altered. This neighborhood is larger than array-swap-index, as evidenced by the fact that $\langle i, j \rangle$ and $\langle j, i \rangle$ are distinct moves as long as $|i - j| > 1$. To make the neighborhood unique, a quotient sort that groups $\langle i, i + 1 \rangle$ and $\langle i + 1, i \rangle$ into an equivalence class should be used. If the size of the range bag is n , each array has $n(n - 2)$ neighbors.

Applications for array-insert-element are the same as for array-swap-index and array-swap-adjacent-index. Possibilities for tabu rules for array-insert-element are discussed in Section 7.1.2.5.

A.12 Ring-Swap-Adjacent-Index

Ring-swap-adjacent-index is a perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements, taken as a ring (successor of last element is first element). This is a minor variant of array-swap-adjacent-index that defines one additional swap. If the size of the range bag is n , each array has n neighbors.

$$\begin{aligned}
 D &\mapsto Bag(\alpha) \\
 I &\mapsto \lambda(R) true \\
 R &\mapsto Array(\alpha) \\
 O &\mapsto \lambda(R, A) (range\_bag(A) = R) \\
 C &\mapsto Nat \\
 N\_info &\mapsto \lambda(R, A, i) (i < size(R)) \\
 N\_is &\mapsto \lambda(R, A, i, new\_A) \\
 &\quad (new\_A = swap\_index(A, i, (i + 1) \bmod size(R)))
 \end{aligned}$$

Figure 158. Ring-Swap-Adjacent-Index

A.13 Ring-Reverse-Subarray

Ring-reverse-subarray is another perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements, taken as a ring (successor of last element is first element). A subarray is identified by a pair of distinct indices, $\langle i, j \rangle$, indicating the elements at positions from i to j , inclusively, and wrapping around the end of the array if $j < i$. Operations for reversing and concatenating arrays are assumed. This neighborhood is larger than array-swap-index: an array of size n has $n(n - 1)$ neighbors. Swapping elements at positions i and j (where $i < j - 2$) can be accomplished in two moves, by reversing the subarray from i to j and then reversing the subarray from $i + 1$ to $j - 1$.

```

D  ↦  Bag( $\alpha$ )
I  ↦   $\lambda(R) \text{ true}$ 
R  ↦  Array( $\alpha$ )
O  ↦   $\lambda(R, A) (\text{range-bag}(A) = R)$ 
C  ↦  Nat, Nat
N_info ↦  $\lambda(R, A, \langle i, j \rangle)$ 
           ( $i < \text{size}(R) \wedge j < \text{size}(R) \wedge i \neq j$ )
N_is  ↦  $\lambda(R, A, \langle i, j \rangle, \text{new\_A})$ 
           ( $\text{new\_A} = \text{ring-reverse-subarray}(A, i, j)$ )

% Reverse subarray A(i..j), wrapping as necessary.
op ring-reverse-subarray : Array, Nat, Nat → Array
definition of ring-reverse-subarray is
  axiom  $\forall(A : \text{Array}, i : \text{Nat}, j : \text{Nat})$ 
    ( $i \leq j \Rightarrow \text{ring-reverse-subarray}(A, i, j)$ 
      = concat(concat(subarray(A, zero, i - 1),
        reverse(subarray(A, i, j))),
        subarray(A, j + 1, size(A) - 1)))
  axiom  $\forall(A : \text{Array}, i : \text{Nat}, j : \text{Nat})$ 
    ( $j < i \Rightarrow \text{ring-reverse-subarray}(A, i, j)$ 
      = concat(concat(reverse(subarray(A, i, size(A) - 1)),
        subarray(A, j + 1, i - 1)),
        reverse(subarray(A, zero, j))))
end-definition

```

Figure 159. Ring-Reverse-Subarray

For a symmetric traveling salesman tour, reversing a subarray is equivalent to dropping two edges and adding two others (48), an instance of k -subset-2-exchange tightened to maintain feasibility. Other sequencing and ordering applications might profitably use this neighborhood, especially if the "edge" view of the array is significant in problem domain terms: that is, if adjacency of elements is more important than absolute position. Each move is its own inverse, but forbidding inverse moves does not guarantee solution cycle prevention. Tabu rules based on adjacency are likely to be the effective ones, in effect forbidding added edges from being dropped and/or dropped edges from being added.

A.14 Ring-Move-Subarray

Ring-move-subarray is yet another perfect, symmetric neighborhood for searching over arrays containing a specified bag of elements, taken as a ring (successor of last element is first element). A transform variable is a 4-tuple, $\langle i, j, k, rev \rangle$, where i is the starting index of the subarray, j is the length of the subarray, k is the number of index positions to move it, and rev is a flag indicating whether to reverse the subarray during the move. A quotient sort on such tuples would be needed to have a unique neighborhood.

```

D  ↦ Bag( $\alpha$ )
I  ↦  $\lambda(R) true$ 
R  ↦ Array( $\alpha$ )
O  ↦  $\lambda(R, A) (range\_bag(A) = R)$ 
C  ↦ Nat, Pos, Pos, Boolean
N_info ↦  $\lambda(R, A, \langle i, j, k, rev \rangle)$ 
         $(i < size(R) \wedge j < size(R) - 2 \wedge k < size(R) - j - 1)$ 
N_is  ↦  $\lambda(R, A, \langle i, j, k, rev \rangle, new\_A)$ 
        (if  $rev$  then  $new\_A = ring\_move\_subarray(ring\_reverse\_subarray(A, i,$ 
                                                 $(i + j - 1) \bmod size(R)), i, j, k))$ 
        else  $new\_A = ring\_move\_subarray(A, i, j, k)$ )

```

Figure 160. Ring-Move-Subarray

For a symmetric traveling salesman tour, this is the same as k-subset-3-exchange, tightened to maintain feasibility. Effective tabu rules are again most likely those that focus on adjacency.


```

% Move subarray  $A(i..i + j - 1)$  up  $k$  index positions, wrapping as necessary
op ring-move-subarray : Array, Nat, Pos, Pos → Array
definition of ring-move-subarray is
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Pos}, k : \text{Pos})$ 
     $(i + j + k \leq \text{size}(R))$ 
     $\Rightarrow \text{ring-move-subarray}(A, i, j, k)$ 
       $= \text{concat}(\text{concat}(\text{subarray}(A, 0, i - 1),$ 
         $\text{subarray}(A, i + j, i + j + k - 1)),$ 
         $\text{concat}(\text{subarray}(A, i, i + j - 1),$ 
         $\text{subarray}(A, i + j + k, \text{size}(R) - 1))))$ 
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Pos}, k : \text{Pos})$ 
     $(i + j \leq \text{size}(R) \wedge i + k < \text{size}(R) \wedge i + j + k > \text{size}(R))$ 
     $\Rightarrow \text{ring-move-subarray}(A, i, j, k)$ 
       $= \text{concat}(\text{concat}(\text{subarray}(A, \text{size}(R) - k, i + j - 1),$ 
         $\text{subarray}(A, \text{size}(R) - (i + j + k), i - 1)),$ 
         $\text{concat}(\text{subarray}(A, i + j, \text{size}(R) - 1),$ 
         $\text{subarray}(A, 0, j - 1))))$ 
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Pos}, k : \text{Pos})$ 
     $(i + j \leq \text{size}(R) \wedge i + k \geq \text{size}(R))$ 
     $\Rightarrow \text{ring-move-subarray}(A, i, j, k)$ 
       $= \text{concat}(\text{concat}(\text{subarray}(A, j, \text{size}(R) - (i + j + k - 1)),$ 
         $\text{subarray}(A, i, i + j - 1)),$ 
         $\text{concat}(\text{subarray}(A, \text{size}(R) - (i + j + k), i - 1),$ 
         $\text{concat}(\text{subarray}(A, i + j, \text{size}(R) - 1),$ 
         $\text{subarray}(A, 0, j - 1))))$ 
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Pos}, k : \text{Pos})$ 
     $(i + j > \text{size}(R) \wedge i + k < \text{size}(R))$ 
     $\Rightarrow \text{ring-move-subarray}(A, i, j, k)$ 
       $= \text{concat}(\text{concat}(\text{subarray}(A, \text{size}(R) - k, \text{size}(R) - 1),$ 
         $\text{subarray}(A, 0, \text{size}(R) - (i + j - 1))),$ 
         $\text{concat}(\text{subarray}(A, \text{size}(R) - (i + j + k), i - 1),$ 
         $\text{subarray}(A, \text{size}(R) - (i + j), \text{size}(R) - (i + j + k - 1))))$ 
  axiom  $\forall (A : \text{Array}, i : \text{Nat}, j : \text{Pos}, k : \text{Pos})$ 
     $(i + j > \text{size}(R) \wedge i + k \geq \text{size}(R))$ 
     $\Rightarrow \text{ring-move-subarray}(A, i, j, k)$ 
       $= \text{concat}(\text{concat}(\text{subarray}(A, j, \text{size}(R) - (i + j + k - 1),$ 
         $\text{subarray}(A, i, \text{size}(R) - 1)),$ 
         $\text{concat}(\text{subarray}(A, 0, \text{size}(R) - (i + j - 1)),$ 
         $\text{concat}(\text{subarray}(A, \text{size}(R) - (i + j + k), i - 1),$ 
         $\text{subarray}(A, \text{size}(R) - (i + j), j - 1))))$ 
end-definition

```

Figure 161. Ring-Move-Subarray Operation

Appendix B. A Proof About Classification Diagrams

This appendix presents a technical result concerning the classification diagrams introduced in Chapter IV, specifically that these diagrams represent commutative squares in the category of specs and interpretations. Srinivas has raised some questions about the validity and nature of such diagrams (78), which this result dispels (and which he assisted with, in fact). That such diagrams are useful has been amply demonstrated in Chapters IV, V, and VI.

Figure 162 shows a generic classification diagram. It can be viewed simply as a diagram of specs and morphisms, but it has much more structure than this. There are three interpretations arranged horizontally at top, middle and bottom, with interpretation morphisms connecting them. The bottom interpretation morphism is composed entirely of definitional extensions. There are three more interpretations arranged vertically, at left, middle and bottom, again with interpretation morphisms between them, the right one composed of definitional extensions. The result of interest is as follows:

Theorem B.1 *A commutative diagram of the form shown in Figure 162 represents a commutative square of interpretations.*

Proof. We need to prove that all of the interpretations from S to A' are equal. Definitions for equality of interpretations and sequential composition of interpretations can be found in (79). There are four interpretations from S to A' through S -as- A' . Since the diagram as a whole commutes, however, the two paths from A to S -as- A' are equal, as are the two from A' to S -as- A' . Thus these four interpretations are all equal. We will prove that the composition of $S \Rightarrow A$ and $A \Rightarrow A'$ is equal to this interpretation. By symmetry, this result will hold for the composition of $S \Rightarrow T$ and $T \Rightarrow A'$ as well, and the conclusion follows.

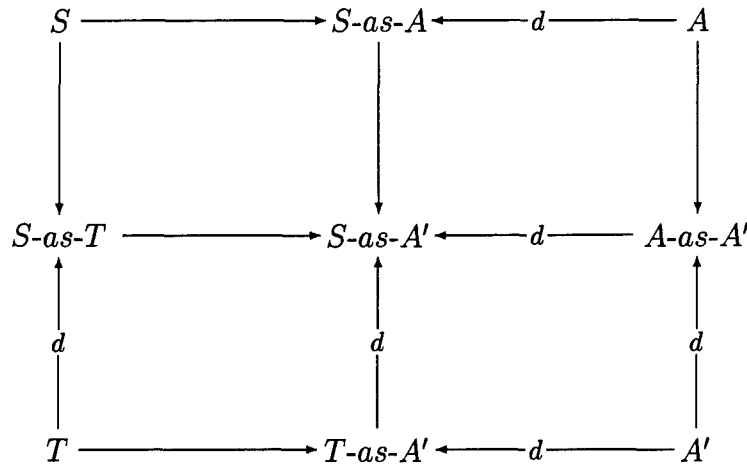


Figure 162. Classification Diagram

Figure 163 shows the top, right and diagonal interpretations of Figure 162 with the arrows labeled¹. We know by assumption that

$$g;h = k;l$$

Compose the top and right interpretations as shown in Figure 164. This yields a pushout object, labeled $S-A-A'$, various new arrows by composition, and a unique arrow p from the pushout to $S-as-A'$; by universality we know

$$g;n = k;o$$

$$l = o;p$$

$$h = n;p$$

We also know p is a definitional extension since if it is not, then $o;p$ is not and therefore l is not, but l is a definitional extension by assumption.

¹The syntax $f;g$ represents a composition of arrows $g \circ f$ written in diagram order. This representation is more convenient when "diagram chasing."

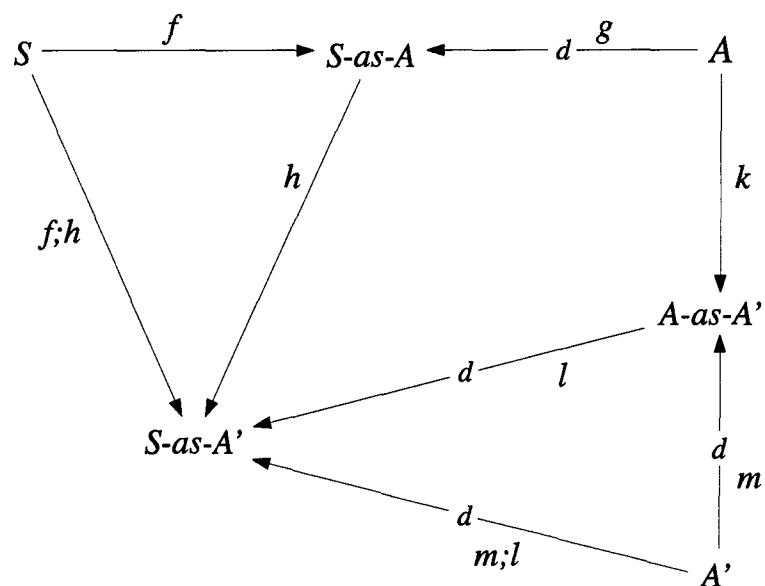


Figure 163. Top, Right and Center Interpretations of Classification Diagram

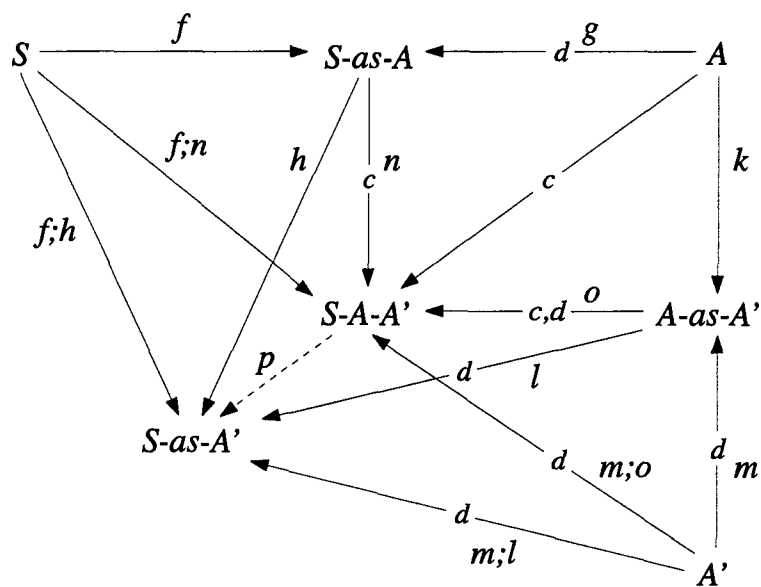


Figure 164. Composition of Top and Right Interpretations

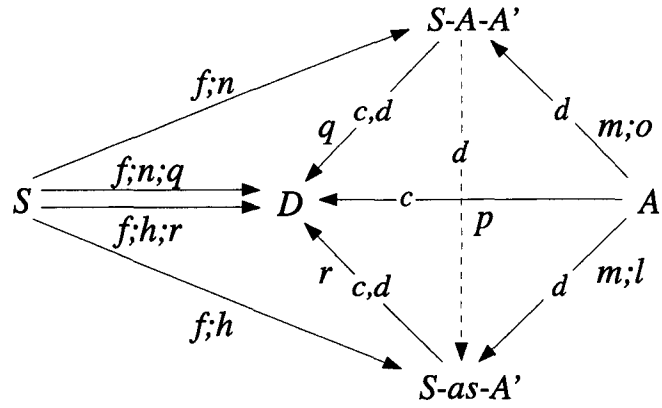


Figure 165. Testing Equality of Diagonal and Composed Interpretations

To show the equality of $S \rightarrow S-A-A' \xleftarrow{d} A'$ and $S \rightarrow S-as-A' \xleftarrow{d} A'$, we again form a pushout, as shown in Figure 165. The pushout spec has been labeled D . The two interpretations are equal if we can prove

$$f;n;q = f;h;r$$

So far we have largely taken an “external” view of the objects and arrows being considered. For the next step we need to look in more detail at what equality of morphisms means in the category **Spec** in particular. The following lemma isolates this part from the rest of the argument.

Lemma. Assume three definitional extensions f , g and p form a commutative triangle as shown in Figure 166, and let (q, r) be the pushout of f and g (only). Then

$$q = p;r$$

Proof. Since morphisms are functions, two morphisms are equal if they map all symbols in the source spec identically to symbols in the target. By this definition, however, $q = p;r$ is not necessarily true: the symbols in C and their p -images in B are both present in the colimit spec $B \oplus_A C$ as separate copies, unless these symbols are in the image of g and f , respectively, and so originate in the shared spec, A .

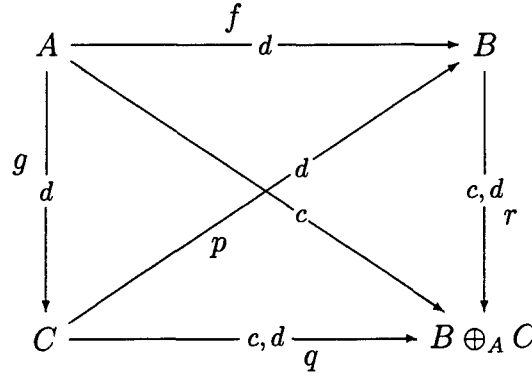


Figure 166. Lemma Diagram

A more general definition of morphism equality given in (79) is that two morphisms $\sigma, \tau : S \rightarrow T$ are equal if

1. For each sort s of S , $\sigma(s) = \tau(s)$
2. For each operation f of S , $\forall(x) \sigma(f)(x) = \tau(f)(x)$.

This definition includes the previous one as a special case. We can extend this definition further by changing the first condition to require only that $\sigma(s)$ and $\tau(s)$ be isomorphic objects in the topos generated by T . Isomorphism of sorts is a well-founded formal notion, and while SPECWARE does not currently support isomorphic sorts, future versions may (78). Using this definition of morphism equality, we can achieve the desired result by proving four separate cases.

Case 1. Let h be an operation symbol in C . If h is in the image of g , then there is some symbol h' in A such that $g(h') = h$. Since g is monic, h' is uniquely determined. Thus we have

$$\begin{aligned}
 q(h) &= (g; q)(h') && h = g(h') \text{ by assumption} \\
 &= (f; r)(h') && \text{pushout squares commute} \\
 &= (g; p; r)(h') && f = g; p \text{ by assumption} \\
 &= (p; r)(h) && h = g(h') \text{ by assumption}
 \end{aligned}$$

Thus for all operation symbols in the range of g , $q = p; r$.

Case 2. Consider now an operation symbol h in C that is not in the image of g . Assume h is mapped to a symbol k in B , which is in turn mapped to a symbol k' in $B \oplus_A C$. All the axioms concerning h in C are translated by p to theorems about k in B , and these theorems are translated by r to theorems about k' in $B \oplus_A C$. Similarly, if q maps h to some symbol k'' in $B \oplus_A C$ then again all the axioms in B concerning h are translated to theorems about k'' . Thus k' and k'' both have all the properties defined in B for h . Since g is a definitional extension, h is fully defined—it has a provably unique value for every input. Thus k' and k'' are equal over their whole domain and satisfy part 2 of the definition above for morphism equality. Thus for all operation symbols of C , $q = p; r$.

Cases 3 and 4. Now consider a sort symbol s in C . If s is in the image of g , then as before $q(s) = (p; r)(s)$. If s is not in the image of g , then there is a sort-axiom defining s in terms of symbols that are in the image of g or are themselves defined in C . As before, s is translated by $p; r$ into some symbol t' in $B \oplus_A C$ and by q into some symbol t'' . The sort axiom of C and the definitions it uses are likewise translated, the latter into theorems. A simple inductive argument starting from sort-axioms and definitions involving only symbols from g and building up to the translated sort-axiom for s shows that the sorts t' and t'' are isomorphic. Thus for all sort symbols of C , $q = p; r$. □

Now we can complete the proof of the theorem.

$$\begin{aligned} f; n; q &= f; n; p; r && \text{because } q = p; r \text{ by the lemma} \\ &= f; h; r && \text{because } n; p = h \text{ by universality} \end{aligned}$$

□

Appendix C. Lowry's Theory of Basic Neighborhoods

Lowry has proposed a theory describing a class of neighborhoods he calls *basic neighborhoods* (51). He offers it as a formal model for a large class of neighborhoods widely used in local search. This appendix describes basic neighborhood theory, evaluates how well it accounts for the library of neighborhoods in Appendix A, and offers some speculations on how the theory might be used to support the creation or discovery of new neighborhoods.

C.1 Basic Neighborhood Theory

Papadimitriou and Steiglitz associate the neighborhood structures used in local search with the notion of a *natural perturbation* of a solution (59). A perturbation is a small, localized change to a solution that transforms it into another. Combinatorial problems in particular often have solution spaces generated by the possible arrangements of a finite set of components according to a set of constraints, and neighborhood structures can be based on local rearrangements of these components (51). Lowry's theory of basic neighborhoods is an attempt to formalize the concept of natural perturbation. He does this in terms of group theory, in particular the theory of group action (a fuller explanation of group action than will be given here can be found in (54)).

The groups Lowry uses are permutation groups, that is, groups consisting of all permutations of a set. The group action shows how a permutation is applied to a solution to generate a new solution. The axioms of group action insure a consistent result: the action of the identity permutation must be to leave the solution unaltered, and the effect of applying two permutations in succession must be the same as the effect of composing the permutations and applying the result. The set used and the effect of the action depend on the data structure involved. If an array type is used, for example, then permuting the set of indices has the effect of altering the order of the elements of the array. If a map type is used, then permuting the domain of the map rearranges the range elements assigned to the affected domain elements. For a map, the range set might be

permuted instead. If a solution has the form of a subset of some base set, then permuting the base set changes the contents of the current subset. For example, if $S = \{1, 2, 3, 4, 5, 6\}$ is the base set, $A = \{1, 2, 3\}$ is the current solution, and $\sigma = (1\ 3\ 5)$ is a permutation (written in cyclic notation, so that 1 is mapped to 3, 3 to 5, 5 to 1 and 2, 4, and 6 are each mapped to themselves), then the result of applying σ to A is $A' = \{2, 3, 5\}$.

Use of a permutation group provides us with an intensional description of a neighborhood without reference to the data of a specific problem instance or current solution. If the set permuted has n elements, however, then the permutation group has $n!$ elements, a number too large for practical use. Instead we use a subset of the full permutation group that constitutes a set of *generators*, which means that the closure of the subset under functional composition is the entire permutation group. Generator sets are used to reduce the size of a neighborhood while maintaining reachability. Lowry proposes the exclusive use of transpositions, meaning permutations that exchange two elements and map the rest to themselves. For arrays, this corresponds to swapping the positions of any two elements, generating the neighborhood we have been calling array-swap-index. Elements of the selected generator set define the binary relation N , while the full permutation group generated corresponds to N^* .

Given a group action, it is necessary to characterize the properties of solutions that it holds invariant. A permutation is a rearrangement that otherwise does not affect the pieces of a solution. For example, the permutations of an array affect only the order of the elements in the array, not their identity or the length of the array. Basic neighborhoods are deliberately general, that is, their invariants are relatively weak in that they typically describe only the structural constraints of the data structures they work on. Consequently, they define large solution spaces that for most problems would include infeasible solutions. Both Lowry's neighborhood tactic and the one described in Chapter VI work by specializing neighborhoods to particular problems, so this characteristic of basic neighborhoods is an advantage. If the exact invariant is difficult to describe, a weaker or

D	$\mapsto$	$datatype1$
I	$\mapsto$	$\lambda(x : D) P(x)$
R	$\mapsto$	$datatype2$
O	$\mapsto$	$\lambda(x : D, z : R) Invariant(x, z)$
C	$\mapsto$	α, α
N_info	$\mapsto$	$\lambda(x : D, z : R, i : \alpha, j : \alpha) (i \in F(x) \wedge j \in F(x))$
N_is	$\mapsto$	$\lambda(x : D, z : R, i : \alpha, j : \alpha, z' : R) (z' = (x, z, i, j))$

Figure 167. Mapping from Abstract Local Search Theory to Basic Neighborhood Theory

stronger condition can be used and the resulting neighborhood theory would be either unreachable or infeasible, respectively. It is also necessary to define the “input data” of a group action, which is data needed either to determine the contents of the set being permuted, or in describing the invariant.

According to Lowry, then, the complete specification of a basic neighborhood consists of a domain sort, *datatype1*, an input condition, *P*, and a range sort, *datatype2*, as for a problem specification; an operation, *F* that computes the set being permuted; a sort, α , for the elements of this set; the *Action* operation; and the invariant maintained by the action, called *Invariant*. The axioms are those of permutation group theory and group action, including the generation condition that assures reachability, and an axiom that the group action maintains the invariant. The domain and range sorts and the input and invariant conditions can be considered a problem specification, as long as one assumes that the invariant is exactly characterized, which Lowry does. Under this assumption, a perfect neighborhood is defined: maintaining the invariant constitutes feasibility and generating the full permutation group constitutes reachability. These neighborhoods are also symmetric. The mapping from spec *Neighborhood* to the elements of this theory is given in Figure 167.

C.2 Evaluation of Basic Neighborhood Theory

The first question to ask about Lowry's theory concerns its generality: does it truly describe the majority of the neighborhoods used in local search algorithms? The second question is, how completely does it describe these neighborhoods, or, are there other considerations that go into designing neighborhood theories. To address these questions, we will consider how well the theory accounts for the neighborhoods defined in Appendix A.

A near-perfect fit is achieved for the array-swap-index neighborhood. As stated earlier, the set being permuted is the set of indices, which is a range of integers, and the action of transposition $(i\ j)$ on an array A is to swap the positions of the elements $A(i)$ and $A(j)$. In cyclic notation the expressions $(i\ j)$ and $(j\ i)$ represent the same transposition, so to achieve uniqueness one must alter the representation, for example by using a quotient sort; this problem was already observed for array-swap-index in Chapter VII. It is interesting to note that an array of length n has $n!$ permutations, the same number as the permutations of the index set. This does not hold true for other neighborhoods.

Array-swap-adjacent-index exploits structure of the index set of an array that is not common to all sets, namely the ordering of the integers. This ordering permits us to define the concept of adjacency and so restrict a basic neighborhood by a supplementary criterion. A consequence is that *N_info* no longer has the form of a test for set membership. This neighborhood can be viewed as an instance of using a generator set other than transpositions, but the dependence of this choice on a property outside the domain of permutations and group action gives us the first indication that basic neighborhood theory does not completely account for all features of even simple neighbors. Ring-swap-adjacent-index provides another example by exploiting a slightly different property of integer ranges.

Accommodating the k -subset-1-exchange neighborhood into basic neighborhood theory requires some additional work. The set being permuted is the base set S itself; the action of trans-

position $(i\ j)$ is to move element i to where j is and to move j to where i is. A neighborhood so defined would permit wasted moves, however, since if i and j are both in the current solution, or if both are excluded from the current solution, then the transposition has no effect. Reflexivity in general is not a desirable characteristic in a local search neighborhood, and in this case is aggravated by a representation for moves that is far from satisfying the uniqueness property. The actual k -subset-1-exchange neighborhood specified in Appendix A can be seen as a refinement of the basic neighborhood that achieves uniqueness and irreflexivity by assigning different roles to i and j , requiring i to be excluded from and j included in the current solution. Assigning these roles breaks the symmetry between $(i\ j)$ and $(j\ i)$ in transpositions, and in so doing solves the general uniqueness problem of equivalence. It is interesting to note that there are only

$$\binom{n}{k} \ll n!$$

k -subsets of an n -element set, suggesting a great deal of redundancy in the group action relating permutations to solutions (that is, many permutations will have an identical effect on a given solution). The inefficiency of a straightforward application of basic neighborhood theory to k -subsets is one effect of this redundancy.

The free-subset neighborhood is even harder to fit into the proposed framework. Here the position of an element i is "toggled" by a move: if present in the current solution it is removed, and if not it is added. This has the quality of exchange common to transposition neighborhoods, but what set is being permuted? One could posit the existence of a "ghost" element that is required to participate in every move, and define the ghost as being both in and not in the current solution (or use two ghosts). Less ghoulishly, one could introduce the notion of *local action*, where the group action has as arguments both a group element (i.e., a permutation) and some additional *locality information* restricting the scope of the action. For free-subset, let the permuted set be $\{0, 1\}$ (or any other 2-element set) and the locality information be the element to move. The action of the (unique) transposition $(0\ 1)$ localized to i is defined so as to move i as desired. Since there

is only one transposition, there is no need to represent it explicitly. This analysis is strained and unsatisfying, however.

Maps have two obvious sets associated with them: their domain and their range (or more generally, their codomain). Permuting the domain set and defining the action of $(i\ j)$ on a map M as setting M at i to $M(j)$ and M at j to $M(i)$ yields the map-swap-domain neighborhood. Permuting the range set and defining the action of $(i\ j)$ on a map M as setting all domain elements current mapped to i to j and all elements mapped to j to i yields the map-swap-range neighborhood. Both of these neighborhoods are well accounted for by basic neighborhood theory, and appear in Lowry's original library of 4 neighborhoods.

Map-set-var-to-val is another difficult case. One can conceive of localizing map-swap-range to a single domain element, so that the swap changes the map at only one point. Thus the effect of a transform (x, i, j) is to modify a map by assigning to variable x the value j if it is currently i , and i if it is currently j . If x is neither i nor j , the transposition has no effect. To avoid such moves, i and j can be assigned roles of current value and new value, respectively, and *N_info* can be modified to check that x is currently assigned the value i . This representation can be further optimized to drop i entirely and the corresponding test, yielding the neighborhood in Appendix A. The effort required to view this neighborhood as a derivative of a basic neighborhood seems excessive, and the number of additional concerns and considerations involved makes the value added by this view seem very small.

The ring-reverse-subarray neighborhood ranges over the same solution space as array-swap-index but is a distinct neighborhood. The set permuted and the action of the group on sequences is the same; what differs is the set of generators chosen. The ordering on integers is exploited to produce a compact intensional representation for "reversing" sequences of integers. The effect is intuitive for sequences viewed as rings and would apply equally well to maps over ordered domains. An attempt to apply this technique to a k -subset problem would suffer from the inefficiencies

encountered with transpositions and so would require further refinement. The ring-move-subarray neighborhood can be similarly analyzed.

Independent-set is a neighborhood for searching over the independent sets of an undirected graph. It is related to free-subset and k -subset-1-exchange, but has many features peculiar to graphs. A transform variable is a single node, which is added to the current solution. As a result, zero or more nodes are removed so that the resulting solution is an independent set. One can view this as an exchange, but the exchange is governed by comparatively tight constraints that are graph-theoretic in nature. That is, the constraints are domain-dependent, and to a much greater degree than we have seen previously, such as with array-swap-adjacent-index. The invariant of this neighborhood is hard to characterize, as we saw in Appendix A. The neighborhood is feasible for general independent sets and reachable for maximal independent sets, but perfect for neither. Lowry's theory makes no explicit provision for imperfect library neighborhoods; our approach allows, and even demands, an exact delineation of the properties exhibited by library neighborhoods, in any combination.

A final example considers a neighborhood not presented in Appendix A. The shape of a binary tree can be altered by "rotating" part of it around a given node, as shown in Figure 168. This rotation holds invariant the in-order traversal of the tree (and hence its range, meaning the data associated with the nodes) and allows a tree of any shape to be transformed into a tree of any other shape. Given as input a sequence of elements (of an unspecified sort), this neighborhood is perfect over all binary trees whose in-order traversal matches the input sequence. We are unaware of this neighborhood being used in a local search algorithm, but it does have application in tree balancing algorithms (8). To date, a way of defining a set whose permutations can act on a tree to produce the effect of this neighborhood has completely eluded us.

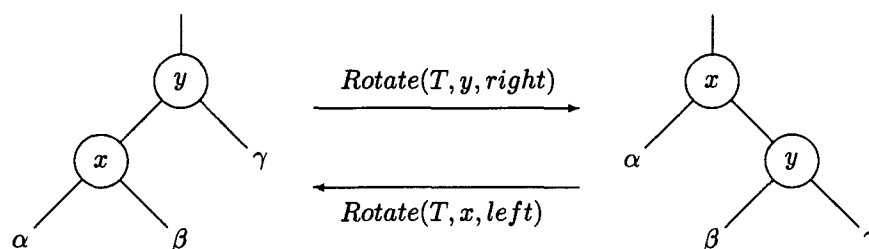


Figure 168. Binary Tree Rotation

C.3 Conclusions

In summary, the “twists” needed to see some neighborhoods as instances of basic neighborhoods are strained and artificial. As it stands, perhaps only half of the neighborhoods in Appendix A comfortably fit into Lowry’s proposed framework. This is a significant fraction, but clearly points out the need for a more general approach to neighborhood structure. Moreover, the optimizations needed to make a good neighborhood are substantial: basic neighborhood theory explains a relatively small part of the structure of some of the neighborhoods in the library. Optimizations address the desirability of uniqueness and efficiency of a neighborhood, and can affect the form of the transform variables as well as the relations *N_info* and *N_is*. These can be seen as refinements of basic neighborhood theory. The opportunities presented by domain-specific properties of particular problems and data types, on the other hand, fundamentally lie outside the scope of Lowry’s theory, and some neighborhoods simply seem to be of other types.

The creative part in applying local search to a problem is first in choosing a suitable representation for solutions in terms of some data structure and second in defining one or more neighborhoods for the data structure that match the constraints of the problem as closely as possible. A fundamental problem with Lowry’s theory is that it is descriptive, not constructive. That is, it describes a class of neighborhoods without immediately suggesting any systematic procedure for generating a basic neighborhood for a given data structure. The discussion above identifies several orthogonal issues in neighborhood design that might eventually lead to a design tactic of sorts for

constructing basic neighborhoods. The following ideas and open questions are listed as possible topics of future research:

- Given a data structure, is there a systematic way of discovering candidate sets for permutation and identifying potential actions for them? For example, a consideration of the constructors or observers of the type might be used.
- Given a permutation group and its action, can its invariant be derived by an automated theorem prover?
- Commonly used sets of generators for permutation groups, such as transpositions and reversals, could be stored in a knowledge base. The choice of a set of generators seems to be independent of other neighborhood design issues (that is, any choice will produce a valid neighborhood; its effectiveness for a given problem is another issue entirely).
- Are uniqueness and efficiency, suitably formalized, strong enough to drive an automated refinement of a basic neighborhood?

Bibliography

1. Aarts, E. H. L., et al. "A Computational Study of Local Search Algorithms for Job Shop Scheduling," *ORSA Journal on Computing*, 6(2):118-125 (Spring 1994).
2. Ahuja, Ravindra, et al. *Networks Flows: Theory, Algorithms and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1993.
3. Ausiello, G. and M. Protasi. "Local Search, Reducibility and Approximability of NP-optimization Problems," *Information Processing Letters*, 54(2):73-79 (Apr 1995).
4. Bailor, Paul. "CSCE 793 class notes." Air Force Institute of Technology, 1994.
5. Bazaraa, Mohhtar S., et al. *Linear Programming and Network Flows* (Second Edition). John Wiley & Sons, 1990.
6. Blaine, Lee and Allen Goldberg. "DTRE—A Semi-Automatic Transformation System." *Constructing Programs from Specifications* edited by B. Möller, 165-203, Elsevier Science Publishers B.V. (North-Holland), 1991.
7. Cai, Jiazhen and Robert Paige. "Program Derivation by Fixed Point Computation," *Science of Computer Programming*, 11:197-261 (1989).
8. Cormen, Thomas H., et al. *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1990.
9. Costa, Daniel. "On the use of some known methods for T -colorings of graphs," *Annals of Operations Research*, 343-358 (1993).
10. Davis, Lawrence. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
11. Della Croce, Federico. "Generalized Pairwise Interchanges and Machine Scheduling," *European Journal of Operational Research*, 83(2) (Jun 1995).
12. Dell'Amico, Mauro and Marco Trubian. "Applying Tabu Search to the Job-shop Scheduling Problem," *Annals of Operations Research*, 41:231 (1993).
13. DeLoach, Scott A. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specifications*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, June 1996.
14. Dowsland, Kathryn A. "Simulated Annealing." *Modern Heuristic Techniques for Combinatorial Problems* edited by Colin R. Reeves, chapter 3, New York: John Wiley & Sons, Inc., 1993.
15. Dunlop, A. E. and Brian W. Kernighan. "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Transactions on Computer-Aided Design*, 4:92-98 (1985).
16. Ehrig, Hartmut and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, 6. EATCS Monographs on Theoretical Computer Science. Berlin; New York: Springer-Verlag, 1985.
17. Ehrig, Hartmut and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, 21. EATCS Monographs on Theoretical Computer Science. Berlin; New York: Springer-Verlag, 1990.
18. Fiduccia, C. M. and R. M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions." *Proceedings of the 19th Design Automation Conference*. 175-181. 1982.
19. French, Simon. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*. West Sussex, England: Ellis Horwood Limited, 1982.

20. Friden, C., et al. "TABARIS: An exact algorithm based on tabu search for finding a maximum independent set in a graph," *Computers and Operations Research*, 17(5):437-445 (1990).
21. Gendreau, Michel, et al. "Solving the Maximum Clique Problem Using a Tabu Search Approach," *Annals of Operations Research*, 41(1/4):385-403 (1993).
22. Gilham, Li-Mei, et al. "Toward Reliable Reactive Systems." *Proceedings of the 5th International Workshop on Software Specification and Design*. May 1989.
23. Glover, Fred. "Tabu search—part I," *ORSA Journal on Computing*, 1(3):190-206 (1989).
24. Glover, Fred. "Tabu search—part II," *ORSA Journal on Computing*, 2(1):4-32 (1990).
25. Glover, Fred. "Tabu search: a tutorial," *Interfaces*, 20(4):74-94 (1990).
26. Glover, Fred, et al. "The Threshold Assignment Algorithm," *Mathematical Programming Study*, 26:12-37 (1986).
27. Glover, Fred and Manuel Laguna. "Tabu Search." *Modern Heuristic Techniques for Combinatorial Problems* edited by Colin R. Reeves, chapter 3, New York: John Wiley & Sons, Inc., 1993.
28. Glover, Fred, et al. "A user's guide to tabu search," *Annals of Operations Research*, 41:3-28 (1993).
29. Goguen, J. A., et al. "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types." *Current Trends in Programming Methodology, Vol. 4: Data Structuring* edited by R. Yeh, Englewood Cliffs, NJ: Prentice-Hall, 1978.
30. Goguen, Joseph A. and Timothy Winkler. *Introducing OBJ3*. Technical Report, 333 Ravenswood Ave, Menlo Park, CA: Computer Science Laboratory SRI International, August 1988.
31. Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
32. Goldblatt, Robert. *Topoi: The Categorical Analysis of Logic*, 98. Studies in Logic and the Foundations of Mathematics. North-Holland, 1984.
33. Grover, L. K. "Local Search and the Local Structure of NP-complete Problems," *Operations Research Letters*, 12(4):235-243 (Oct 1992).
34. Guttag, John and James Horning. *Larch: Languages and Tools for Formal Specification*. New York, NY: Springer-Verlag New York, Inc., 1993.
35. Hansen, P. "The Steepest Ascent Mildest Descent Heuristic for Combinatorial Programming." *Congress on Numerical Methods in Combinatorial Optimization*. 1986. Capri, Italy.
36. Hansen, P. and B. Jaumard. "Algorithms for the Maximum Satisfiability Problem," *Computing*, 44:279-303 (1990).
37. Hartrum, Thomas C. and Paul D. Bailor. "Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies." *Software Engineering Education Proceedings*. Pittsburgh, PA: Software Engineering Institute (SEI), January 1994.
38. Hertz, A. "Tabu Search for Large Scale Timetabling Problems," *European Journal of Operational Research*, 54:39-47 (1991).
39. Hooker, J. N. "Resolution vs. Cutting Plane Solution of Inference Problems: Some Computational Experience," *Operations Research Letters*, 7(1):1-7 (Feb 1988).
40. Johnson, David S., et al. "Optimization by simulated annealing: an experimental evaluation; Part I, graph partitioning," *Operations Research*, 37(6):865-892 (1989).

41. Johnson, David S., et al. "Optimization by simulated annealing: an experimental evaluation; Part II, graph coloring and number partitioning," *Operations Research*, 39(3):378-406 (1991).
42. Johnson, David S., et al. "How Easy is Local Search?," *Journal of Computer and System Sciences*, 37:79-100 (1988).
43. Jüllig, Richard K. and Yellamraju V. Srinivas. "Diagrams for Software Synthesis." *Proceedings of the Eighth Knowledge-Based Software Engineering Conference*. 10-19. Sep 1993.
44. Kernighan, Brian W. and Shen Lin. "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, 49(2):291-307 (Feb 1970).
45. Lasdon, Leon S. *Optimization Theory for Large Systems*. MacMillan Publishing Company, 1970.
46. Liepins, G. E. and M. R. Hilliard. "Genetic algorithms: foundations and applications," *Annals of Operations Research*, 21:31-58 (1989).
47. Liepins, G. E., et al. "Genetic algorithms applications to set covering and traveling salesman problems." *Operations Research and Artificial Intelligence: The integration of Problem-Solving Strategies* edited by D. E. Brown and C. C. White, 29-57, Kluwer, Boston, 1990.
48. Lin, Shen. "Computer Solutions of the Traveling Salesman Problem," *The Bell System Technical Journal*, 44(10):2245 - 2269 (December 1965).
49. Lin, Shen and Brian W. Kernighan. "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research*, 21:498 - 516 (1973).
50. Lowry, Michael R. *Algorithm Synthesis Through Problem Reformulation*. PhD dissertation, Stanford University, 1989.
51. Lowry, Michael R. "Automating the Design of Local Search Algorithms." *Automating Software Design* edited by Michael R. Lowry and Robert D. McCartney, 515-546, Menlo Park, CA: AAAI Press, 1991.
52. Lubars, Mitchell D. "The ROSE-2 Strategies for Supporting High-Level Software Design Reuse." *Automating Software Design* edited by Michael R. Lowry and Robert D. McCartney, 93-118, Menlo Park, CA: AAAI Press, 1991.
53. Mac Lane, Saunders. *Categories for the Working Mathematician*. New York: Springer-Verlag, 1972.
54. Mac Lane, Saunders and Garrett Birkhoff. *Algebra*. New York, NY: Chelsea Publishing Company, 1993.
55. Manna, Zohar and Richard Waldinger. "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering*, 18(8):674-704 (Aug 1992).
56. Martin, O., et al. "Large-step Markov Chains for the TSP incorporating Local Search Heuristics," *Operations Research Letters*, 11(4):219-224 (May 1992).
57. McCartney, Robert D. "Subtask Independence and Algorithm Synthesis." *Automating Software Design* edited by Michael R. Lowry and Robert D. McCartney, 315-338, Menlo Park, CA: AAAI Press, 1991.
58. Mitchell, David, et al. "Hard and Easy Distributions of SAT Problems." *Proceedings of the Tenth National Conference on Artificial Intelligence*. 459-465. Menlo Park, CA: AAAI Press, 1992.
59. Papadimitriou, C. H. and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

60. Potter, B., et al. *An Introduction to Formal Specification and Z*. New York: Prentice Hall, 1991.
61. Reasoning Systems, Palo Alto, CA. *Software Refinery Training Manual*, 1991.
62. Reubenstein, Howard B. "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Transactions on Software Engineering*, 17(3) (Mar 1991).
63. Rich, Charles. "A Formal Representation for Plans in the Programmer's Apprentice." *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*. 1981.
64. Selman, Bart, et al. "A New Method for Solving Hard Satisfiability Problems." *Proceedings of the Tenth National Conference on Artificial Intelligence*. 440-446. Menlo Park, CA: AAAI Press, 1992.
65. Setliff, Dorothy. "On the Automatic Selection of Data Structure and Algorithms." *Automating Software Design* edited by Michael R. Lowry and Robert D. McCartney, 207-226, Menlo Park, CA: AAAI Press, 1991.
66. Sinclair, Marius. "Comparison of the Performance of Modern Heuristics for Combinatorial Optimization on Real Data," *Computers and Operations Research*, 20(7):687-695 (1993).
67. Skorin-Kapov, Jadranka. "Tabu Search Applied to the Quadratic Assignment Problem," *ORSA Journal on Computing*, 2(1):33-45 (Winter 1990).
68. Smith, Douglas R. "Applications of a Strategy for Designing Divide-and-Conquer Algorithms," *Science of Computer Programming*, 8:213-229 (1987).
69. Smith, Douglas R. "KIDS—A Semi-automatic Program Development System," *IEEE Transactions on Software Engineering*, 16(9):1024-1043 (September 1990).
70. Smith, Douglas R. "Structure and Design of Problem Reduction Generators." *Constructing Programs from Specifications* edited by B. Möller, 91 - 123, Elsevier Science Publishers B.V. (North-Holland), 1991.
71. Smith, Douglas R. *A Classification Approach to Design*. Technical Report KES.U.93.4, Palo Alto, CA: Kestrel Institute, Nov 1993.
72. Smith, Douglas R. "Constructing Specification Morphisms," *Journal of Symbolic Computation*, 15(5-6):571-606 (1993).
73. Smith, Douglas R. and Michael R. Lowry. "Algorithm Theories and Design Tactics," *Science of Computer Programming*, 14:305-321 (1990).
74. Smith, Douglas R. and Eduardo A. Parra. "Transformational Approach to Transportation Scheduling." *Proceedings of the Eight Knowledge-Based Software Engineering Conference, Chicago, IL*. September 1993.
75. Smith, Douglas R., et al. *Synthesis of High-Performance Transportation Schedulers*. Technical Report KES.U.95.1, Palo Alto, CA: Kestrel Institute, Mar 1995.
76. Srinivas, Yellamraju V. *Algebraic Specification: Syntax, Semantics, Structure*. Technical Report 90-15, Palo Alto, CA: Kestrel Institute, June 1990.
77. Srinivas, Yellamraju V. *Category Theory: Definitions and Examples*. Technical Report 90-14, Palo Alto, CA: Kestrel Institute, February 1990.
78. Srinivas, Yellamraju V. Personal communication, April 1995.
79. Srinivas, Yellamraju V. and Richard K. Jüllig. *SPECWARE: Formal Support for Composing Software*. Technical Report KES.U.94.5, Palo Alto, CA: Kestrel Institute, Dec 1994.

80. Srinivas, Yellamraju V., et al. *Specware Language Manual, Version 1.2*. Kestrel Institute, Palo Alto, CA, March 1996.
81. Thompson, Gary M. "A simulated-annealing heuristic for shift scheduling using non-continuously available employees," *Computers and Operations Research*, 23(1):275-288 (1996).
82. Tsang, Edward. *Foundations of Constraint Satisfaction*. London, San Diego: Academic Press Limited, 1993.
83. Waters, Richard C. "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transactions on Software Engineering*, 7(11) (Nov 1981).

Vita

Captain Robert P. Graham, Jr. [REDACTED] With the assistance of a 4-year AFROTC scholarship, in June 1986 he received a Bachelor of Science in Computer Science in Honors, *summa cum laude*, from Virginia Polytechnic Institute and State University and was commissioned into the U.S. Air Force. His first assignment was to the Air Force Institute of Technology (AFIT), where he received a Master of Science in Computer Systems in 1988 and was named a Distinguished Graduate. Capt Graham was transferred to the 7th Communications Group, Pentagon, as a computer programmer and analyst. In October 1988 he became a member of the Church of Jesus Christ of Latter-Day Saints and in December of that year was wed to Kathryn Jean Bunker. Capt and Mrs. Graham were sealed for time and all eternity in 1989 at the Arizona temple in Mesa, Arizona. Capt Graham attended Squadron Officer School in residence in 1992. In 1993, Capt Graham returned to AFIT to pursue the Ph.D. Upon graduation, he will remain at AFIT as an assistant professor in the Computer Science and Engineering Division of the Department of Electrical and Computer Engineering, School of Engineering. Capt Graham currently serves as the ward clerk of the Huber Heights Ward of the Church of Jesus Christ of Latter-Day Saints. Captain and Mrs. Graham have three children: Nathaniel (REDACTED) Miranda (REDACTED) and Alexander (REDACTED).

Address: 7210 [REDACTED]
[REDACTED]

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1996		3. REPORT TYPE AND DATES COVERED Doctoral Dissertation
4. TITLE AND SUBTITLE ALGEBRAIC ALGORITHM DESIGN AND LOCAL SEARCH				5. FUNDING NUMBERS
6. AUTHOR(S) Robert P. Graham, Jr., Captain, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2950 P Street WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/DS/ENG/96-10
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. Glenn Durbin NSA/Y21 9800 Savage Rd, Suite 6718 Fort Meade, MD 20755-6718				10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) Formal, mathematically-based techniques promise to play an expanding role in the development and maintenance of the software on which our technological society depends. Algebraic techniques have been applied successfully to algorithm synthesis by the use of algorithm theories and design tactics, an approach pioneered in the Kestrel Interactive Development System (KIDS). An algorithm theory formally characterizes the essential components of a family of algorithms. A design tactic is a specialized procedure for recognizing in a problem specification the structures identified in an algorithm theory and then synthesizing a program. Design tactics are hard to write, however, and much of the knowledge they use is encoded procedurally in idiosyncratic ways. Algebraic methods promise a way to represent algorithm design knowledge declaratively and uniformly. We describe a general method for performing algorithm design that is more purely algebraic than that of KIDS. This method is then applied to local search. Local search is a large and diverse class of algorithms applicable to a wide range of problems; it is both intrinsically important and representative of algorithm design as a whole. A general theory of local search is formalized to describe the basic properties common to all local search algorithms, and applied to several variants of hill climbing and simulated annealing. The general theory is then specialized to describe some more advanced local search techniques, namely tabu search and the Kernighan-Lin heuristic.				
14. SUBJECT TERMS Knowledge-based software engineering, software synthesis, algorithm design, algebraic methods, local search, tabu search, simulated annealing, Kernighan-Lin heuristic, category theory				15. NUMBER OF PAGES 385
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	