INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
NOTE TO USERS

The original manuscript received by UMI contains pages with indistinct print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI
THE SPATIAL AGGREGATION LANGUAGE FOR MODELING AND CONTROLLING DISTRIBUTED PHYSICAL SYSTEMS

DISSERTATION

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

By

Christopher J. Bailey-Kellogg, B.S., M.S.

The Ohio State University

1999

Dissertation Committee:
Feng Zhao, Adviser
Balakrishnan Chandrasekaran
John Josephson

Approved by

Adviser

Department of Computer and Information Science
ABSTRACT

Many important science and engineering applications, such as predicting weather patterns, controlling the temperature distribution over a semiconductor wafer, and controlling the noise of a photocopy machine, require interpreting data and designing decentralized controllers for spatially distributed systems. This thesis describes the Spatial Aggregation Language (SAL), a novel programming language and environment supporting data interpretation and control tasks for distributed physical systems. SAL provides a set of powerful, high-level components that make explicit use of domain-specific physical knowledge, such as metrics, adjacency relations, and equivalence predicates, in order to uncover and exploit structures in distributed physical data at multiple levels of abstraction. The language data types and operators manipulate structured representations of spatial objects in distributed physical systems at multiple levels of abstraction. The programming environment supports rapid prototyping of application programs and interactive manipulation of the resulting structures. In comparison with existing tools, the Spatial Aggregation Language offers high level programming abstractions explicitly encoding physical knowledge; this approach supports a variety of inference, explanation, tutoring, and design tasks.

This thesis presents as a case study novel approaches to decentralized control design, in the context of thermal regulation. This case study develops novel algorithms for control placement and parameter design for systems with large numbers of coupled
variables. These algorithms exploit physical knowledge of locality, linear superpos-
ability, and continuity, encapsulated in influence graphs representing dependencies of
field nodes on control nodes. The control placement design algorithms utilize influ-
ence graphs to decompose a problem domain so as to decouple the resulting regions.
The decentralized control parameter optimization algorithms utilize influence graphs
to efficiently evaluate thermal fields and to explicitly trade off computation, com-
munication, and control quality. By leveraging the physical knowledge encapsulated
in influence graphs, these control design algorithms are more efficient than standard
techniques, and produce designs explainable in terms of problem structures. This
case study demonstrates the utility of the Spatial Aggregation Language operators in
supporting the programming of these computations in a vocabulary natural for the
domain.
To Christine
ACKNOWLEDGMENTS

This dissertation in particular, and my graduate school career in general, have benefited greatly from the contributions of many people. I would especially like to recognize and thank the following people:

First and foremost, my wife, Christine, my family, the Kelloggs, and my wife’s family, the Baileys. I cannot express how much their love and support have meant to me over the years — this would not have been possible without them. I dedicate this thesis to Christine, who has been a constant source of strength during these many months of “just a few more months.” I am incredibly lucky to have such a wonderful partner with whom to share my life.

My adviser, Feng Zhao. His infectious excitement and ideas about intelligent simulation got me started down this path, and his guidance and encouragement have kept me going. I’ve enjoyed working with him — brainstorming, hacking, writing, and presenting — and he has taught me a great deal about how to conduct research. I also appreciate all that he has done behind the scenes, working in my best interests.

My other committee members, JJ and Chandra. I appreciate their help with the dissertation, and have enjoyed interacting with them on other research projects. My many conversations with JJ have influenced me to see instances of abductive reasoning everywhere I turn.
My unofficial committee member, Ken Yip. He has collaborated on Spatial Aggregation and supported my work for many years. His research, which exemplifies the potential of intelligent simulation, serves as a source of inspiration and motivation for me.

The other members of the Intelligent Simulation Group at Ohio State, Xingang Huang, Shiou Loh, Jeff May, and Iván Ordóñez. Our friendship has meant a lot to me, and this research has benefited a great deal from our interesting discussions in group meetings. Iván helped enormously in hashing out much of the language design and implementing many chunks of the language library. Xingang's ideas on correlations and super-/substructure proved key to some of the examples in this document. Jeff and Loh have pursued interesting questions arising from the application of Spatial Aggregation ideas to a working magnetic levitation system.

Various researchers at Xerox PARC (Computation and Matter Area of SPL) with whom I had the privilege of working for the final year of my research. Patrick Cheung, Marcus Fromherz, John Gilbert, Warren Jackson, John Lamping, Rachel Lau, and Mark Yim provided helpful feedback on my work.

This work was supported in part by NSF Young Investigator award CCR-9457802, ONR Young Investigator award N00014-97-1-0599, a Sloan Research Fellowship, NSF grant CCR-9308639, and a Xerox Foundation grant.
VITA

March 8, 1970 .................................................. Born, Abingdon, Virginia

1993 ............................................................... B.S./M.S. Computer Science.
Massachusetts Institute of Technology

1993; 1997 ....................................................... University Fellow.
The Ohio State University

1994–1995 ....................................................... Graduate Teaching Associate.
The Ohio State University

Intelligent Simulation Group.
The Ohio State University

1998 .............................................................. Research Intern,
Xerox Palo Alto Research Center

PUBLICATIONS

Research Publications


Instructional Publications


FIELDS OF STUDY

Major Field: Computer and Information Science
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Challenges</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Significance</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>7</td>
</tr>
<tr>
<td>2. Related Work</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Reasoning about Physical Systems</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Structure Discovery</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Spatial Reasoning</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Engineering and Scientific Computing</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Advanced Programming Languages</td>
<td>16</td>
</tr>
</tbody>
</table>
3. The Spatial Aggregation Language

3.1 Spatial Aggregation Framework
   3.1.1 Field Ontology
   3.1.2 Multi-Level Spatial Aggregates
   3.1.3 Uniform Vocabulary
   3.1.4 Structure-Based Control Design
   3.1.5 Simple Example: Trajectory Bundling

3.2 Spatial Aggregation Language Definition
   3.2.1 Spatial Object
   3.2.2 Compound Objects
   3.2.3 Means of Abstraction
   3.2.4 SAL Programming Elements Summary

3.3 Spatial Aggregation Language Implementation
   3.3.1 SAL C++ Library
   3.3.2 SAL Interpreter

3.4 Example Programs
   3.4.1 Dynamical System Analysis
   3.4.2 Dynamical System Analysis Exploration
   3.4.3 Vector Field Analysis
   3.4.4 Boundary Tracing

3.5 Programming Style

4. Decentralized Control Design

4.1 Introduction
   4.1.1 Problem Domain
   4.1.2 Overview of the Approach

4.2 Modeling Heat Flow

4.3 Influence Graph
   4.3.1 Thermal Hill
   4.3.2 Influence Graph Construction
   4.3.3 Influence Graph Properties
   4.3.4 Alternate Views of Influences

4.4 Control Placement Design
   4.4.1 Control Probes
   4.4.2 Evaluating Control Decoupling
   4.4.3 Evaluating Region Atomicity
   4.4.4 Control Probe Partitioning
   4.4.5 Performance
   4.4.6 Discussion
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Components of the Spatial Aggregation Language.</td>
<td>29</td>
</tr>
<tr>
<td>3.2 Major cell complex operations.</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Major geometric object operations.</td>
<td>42</td>
</tr>
<tr>
<td>3.4 Major space operations.</td>
<td>45</td>
</tr>
<tr>
<td>3.5 Collection-oriented cell complex operations.</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Major field operations.</td>
<td>49</td>
</tr>
<tr>
<td>3.7 Additional major field operations.</td>
<td>50</td>
</tr>
<tr>
<td>3.8 Major neighborhood graph operations.</td>
<td>53</td>
</tr>
<tr>
<td>3.9 Additional major neighborhood graph operations.</td>
<td>54</td>
</tr>
<tr>
<td>3.10 Major classifier operations.</td>
<td>56</td>
</tr>
<tr>
<td>3.11 Major abstractor operations.</td>
<td>60</td>
</tr>
<tr>
<td>3.12 Summary of SAL programming elements.</td>
<td>61</td>
</tr>
<tr>
<td>3.13 Summary of SAL C++ library.</td>
<td>63</td>
</tr>
<tr>
<td>3.14 Examples of SAL interpreter use.</td>
<td>65</td>
</tr>
<tr>
<td>3.15 Core SAL interpreter syntax.</td>
<td>66</td>
</tr>
</tbody>
</table>
3.16 SAL code for first level of trajectory bundling application. .......... 71
3.17 SAL code for second level of trajectory bundling application. .......... 74
3.18 SAL code for computing point neighborhood graphs during exploration of trajectory bundling application. ....................... 75
3.19 SAL code for filtering point neighborhood graph during exploration of trajectory bundling application. ....................... 80
3.20 SAL code for classifying trajectories during exploration of trajectory bundling application. ....................... 83
3.21 SAL code for first part of vector field analysis application. .......... 86
3.22 SAL code for second part of vector field analysis application. .......... 87
3.23 SAL code for first level of boundary tracing application. .............. 92
3.24 SAL code for second level of boundary tracing application. .............. 95
4.1 SAL disaggregate operator. ......................................................... 113
4.2 SAL algorithmic description of relaxation-based heat equation solution. 117
4.3 SAL relax operator. ......................................................... 118
4.4 SAL coarsen and refine operators. ......................................................... 119
4.5 SAL algorithmic description of multi-level solving. ....................... 120
4.6 SAL algorithmic description of domain decomposition-based solving. 123
4.7 SAL algorithmic description of spectral partitioning. ....................... 124
4.8 Primary influence graph operations. ......................................................... 129
4.9 SAL near-field/far-field test operator. ......................................................... 130
4.10 SAL probe operators. ......................................................... 138
4.11 SAL influence similarity operator. ................................................................. 140
4.12 SAL class atomicity operator. ........................................................................ 142
4.13 SAL algorithmic description of probe clustering. ......................................... 143
4.14 Performance data for decomposition quality. .................................................. 147
4.15 Performance data for decomposition-based control design. ....................... 150
4.16 SAL algorithmic description of decentralized optimization algorithm. .... 154
4.17 SAL decentralized optimization operator. ....................................................... 155
4.18 SAL algorithmic description of efficient field evaluation. ............................ 156
4.19 SAL influence-based field update operator. .................................................... 157
4.20 SAL algorithmic description of reduced-communication field evaluation. 159
4.21 SAL supervisor construction operator. ........................................................... 162
4.22 SAL algorithmic description of supervisor optimization. ............................ 163
4.23 Performance data for communication reduction in optimization. .............. 166
4.24 Performance data for cooperative optimization. .......................................... 169
A.1 Spatial object classes. ..................................................................................... 183
A.2 Geometric object classes. ................................................................................ 183
A.3 Space classes. .................................................................................................. 184
A.4 Field classes. .................................................................................................. 184
A.5 Ngraph classes. ............................................................................................... 185
A.6 Classifier classes. ........................................................................................... 186
A.7 Abstractor classes. .......................................................................................... 186

xiv
A.8 Inspector classes. ............................................................. 187
B.1 Op classes. ................................................................. 191
B.2 Syntax for preprocessor-based operation passing. .......... 193
B.3 Operation-passing preprocessor output. ....................... 194
B.4 Example of preprocessor-based operation passing. ........ 195
B.5 Primitive interpreter operations. ................................... 198
B.6 Syntax of input file specifying types to be used in the SAL interpreter. 199
B.7 Syntax of input file specifying functions to be used in the SAL interpreter. 199
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Examples of fields.</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Spatial Aggregation multi-layer spatial aggregates, uncovered by a uniform vocabulary of operators utilizing domain knowledge.</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Spatial Aggregation data flow for trajectory bundling application.</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>Example steps in trajectory bundling application.</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Some spatial objects in a meteorology application.</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>The same topological structure with different coordinates.</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Example cells.</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>Legal cell complexes.</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Illegal cell complexes.</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>Atomic and non-atomic cell complexes.</td>
<td>35</td>
</tr>
<tr>
<td>3.11</td>
<td>The structural relations of a quadrilateral cell complex.</td>
<td>36</td>
</tr>
<tr>
<td>3.12</td>
<td>A 2-D square embedded in a 3-D metric space.</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>Local metric spaces.</td>
<td>40</td>
</tr>
<tr>
<td>3.14</td>
<td>Geometry transformations.</td>
<td>41</td>
</tr>
<tr>
<td>3.15</td>
<td>A space collects spatial objects for element-wise and global operations.</td>
<td>43</td>
</tr>
</tbody>
</table>
3.16 A field associates objects and features. ............................................................. 47
3.17 Ngraphs localize computation with object adjacencies. ................................. 51
3.18 Delaunay neighborhood graph on points builds a cell complex with segments for connected points and triangles for connected segments. 52
3.19 Rectilinear grid supports directional neighbor queries based on geometry of input space. ................................................................. 55
3.20 Efficient classification process. ................................................................. 58
3.21 Examples of abstraction. ........................................................................... 59
3.22 Global/local properties illustrated with the inside-outside problem. ........ 60
3.23 The SAL interpreter in action: interactive code evaluation and graphical inspection of results. ................................................................. 68
3.24 Example steps in first level of SAL implementation of trajectory bundling application. ................................................................. 70
3.25 Example steps in second level of SAL implementation of trajectory bundling application. ................................................................. 73
3.26 Corresponding points of trajectories based on triangulation. .................. 75
3.27 Point neighborhood graphs during exploration of trajectory bundling application. ................................................................. 76
3.28 Potential problems in using k-nearest-neighbors ngraph for trajectory bundling. ................................................................. 78
3.29 Filtered point neighborhood graphs during exploration of trajectory bundling application. ................................................................. 79
3.30 Trajectory aggregation and classification during exploration of trajectory bundling application. ................................................................. 82
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.31</td>
<td>Example steps in first part of SAL implementation of vector field analysis application</td>
<td>85</td>
</tr>
<tr>
<td>3.32</td>
<td>Example steps in second part of SAL implementation of vector field analysis application</td>
<td>88</td>
</tr>
<tr>
<td>3.33</td>
<td>A bitmap containing two overlapping rectangular boundary objects. A boundary tracing operation can separate the two objects.</td>
<td>90</td>
</tr>
<tr>
<td>3.34</td>
<td>Example steps in first level of SAL implementation of boundary tracing application</td>
<td>91</td>
</tr>
<tr>
<td>3.35</td>
<td>Example steps in second level of SAL implementation of boundary tracing application</td>
<td>94</td>
</tr>
<tr>
<td>4.1</td>
<td>Rapid thermal processing system welds by moving a part on a positioning table based on feedback from an infrared camera.</td>
<td>106</td>
</tr>
<tr>
<td>4.2</td>
<td>Rapid thermal processing for semiconductor manufacturing maintains a uniform temperature distribution by independent control to separate rings of heat lamps.</td>
<td>106</td>
</tr>
<tr>
<td>4.3</td>
<td>Industrial heat treatment of a piece of material.</td>
<td>107</td>
</tr>
<tr>
<td>4.4</td>
<td>Overview of SAL-based computational patterns in modeling heat flow.</td>
<td>110</td>
</tr>
<tr>
<td>4.5</td>
<td>Overview of SAL-based decentralized control design for thermal regulation.</td>
<td>111</td>
</tr>
<tr>
<td>4.6</td>
<td>Domain discretizations.</td>
<td>113</td>
</tr>
<tr>
<td>4.7</td>
<td>Neighborhood graphs for diffusion simulation.</td>
<td>116</td>
</tr>
<tr>
<td>4.8</td>
<td>SAL data flow for relaxation-based heat equation solution.</td>
<td>117</td>
</tr>
<tr>
<td>4.9</td>
<td>Multi-level coarsening/refining.</td>
<td>119</td>
</tr>
<tr>
<td>4.10</td>
<td>SAL data flow for multi-level approximation-based solving of the heat equation.</td>
<td>120</td>
</tr>
</tbody>
</table>
4.11 SAL data flow for domain decomposition-based solving of the heat equation ................................................................. 123
4.12 SAL data flow for spectral partitioning .................................................. 125
4.13 Steady-state thermal hills around sources ......................................... 126
4.14 Thermal hill for a source on a complex domain .................................... 126
4.15 An influence hill partitions a field into near and far fields relative to a control .......................................................... 130
4.16 Linear superposability of thermal hills .................................................. 132
4.17 Iso-influence contours for material and heat source .............................. 133
4.18 Gradient vector directions for material and heat source ....................... 133
4.19 Gradient vector trajectories for material and heat source ..................... 134
4.20 Temperature fields exhibit structures in response to heat source probes 136
4.21 Similarity of flows in the far fields of control probes suggests indistinguishability of control placement .......................... 139
4.22 Control probe placement with potential non-atomic decomposition ....... 141
4.23 Overlapping near fields indicate probe class atomicity ........................ 142
4.24 SAL data flow for probe clustering algorithms ...................................... 144
4.25 Probe merging example: probe neighborhood graph ............................ 144
4.26 Probe merging example: influence gradient vectors from two probes .... 145
4.27 Probe merging example: region decomposition after merging ............... 145
4.28 Performance data indicate that the decomposition algorithm supports trading decomposition quality for computation 148

xix
4.29 Performance data indicate that decomposition-based control placement design supports trading control quality for computation.

4.30 SAL data flow for the basic decentralized optimization algorithm.

4.31 SAL data flow for efficient field evaluation.

4.32 Influence structure supports reduced communication between control node and field nodes.

4.33 SAL data flow for reduced communication optimization.

4.34 Ridge problem in control optimization.

4.35 Supervisor control shifts control value from one control to another.

4.36 SAL data flow for joint optimization.

4.37 Performance data indicate that influence graphs support trading optimization quality for amount of communication.

4.38 Performance data indicate that influence graphs support cooperative optimization.

4.39 SAL operators support high-level programs elaborating the consequences of partial differential equations.

4.40 Influence graph operators support decentralized control design.
CHAPTER 1

INTRODUCTION

1.1 Challenges

Many important science and engineering applications require interpreting data and designing decentralized controllers for spatially distributed systems. For example, in order to predict the weather, meteorologists utilize massive amounts of pressure, temperature, and wind velocity data collected from spatially distributed sensors. Similarly, in designing aircraft with minimal drag, engineers study wind tunnel and simulation data specifying airflow over a body at many points and over many instants of time. Recent advances in the fabrication of low-cost, large-scale arrays of micro-electromechanical systems (MEMS) have enabled the construction of "smart matter" systems that integrate sensing, computation, and actuation at a fine grain. Recent MEMS applications include manipulation of parts with distributed arrays of cilia-like actuators [9] and stabilization of beams with piezoelectric materials that sense and counteract buckling [7].

Interpretation and control tasks for distributed physical systems encounter a number of challenges. In lumped parameter systems such as circuits, topology is the most important property. However, in distributed parameter systems, additional spatial properties are also relevant. For example, temperature fields are influenced by the
geometry of the domain, spatial variations in material property, and boundary conditions. The additional complexity makes it much harder to apply analytic methods to determine closed-form solutions: instead, simulation on large-scale discretizations is often required.

Interpretation and control applications for distributed physical systems are limited by physical laws and physical hardware constraints. Sensors and actuators interact predominantly only with local regions of space. As a result, global interpretations must be extracted from collections of locally measured data, and global control laws must be enforced by local actuation rules. The difficulty of constructing local to global mappings and back is compounded by global coupling between local nodes. In addition to strong interactions with local areas of space, sensors and actuators have weaker interactions with other areas of space. For example, a signal to dampen local acoustic noise eventually propagates to other regions and interferes with efforts to dampen noise in those regions. Sensor and control designs must account for such coupling.

Data interpretation applications for spatially distributed systems are often challenged by the massive amount of data, either collected from arrays of sensors or produced by simulations running on fine-grained discretizations. Global methods manipulating entire data sets reach computational limitations as the size of the data sets increases. Instead, applications must rely on local methods that manipulate separate subsets of the data relatively independently. For example, domain decomposition methods [12] for solving partial differential equations (PDEs) form subregions of a discretization and iteratively combine and refine independent solutions for the subregions. Applications can also use data reduction and approximations to reason about
a problem at multiple levels of abstraction. For example, multigrid methods [10] iterate between fine-grained and coarse-grained solutions to PDEs. Meteorologists use abstract structures such as isobars, pressure troughs, and pressure cells to reason about the underlying pressure data at a higher level of abstraction.

What makes it possible to overcome these challenges and design data interpretation and control applications for distributed physical systems? Physical properties such as continuity and locality give rise to regions of uniformity in spatially distributed data. Spatial Aggregation theory [86] proposes a uniform mechanism utilizing explicit representations of such knowledge, expressed as metrics, adjacency relations, and equivalence predicates, to uncover and exploit structures in physical data. Spatial Aggregation follows an imagistic reasoning [87] style, applying vision-like routines to manipulate multi-layer geometric and topological structures in spatially distributed data. Control design applications use similar techniques to navigate through imagistic representations of the effects of control actions [89]. This thesis describes a programming language, the Spatial Aggregation Language, supporting the construction of applications in this style, and a case study using the language to solve a difficult engineering design problem — decentralized control design.

1.2 Contributions

This thesis makes contributions in two areas: the design and implementation of the Spatial Aggregation Language (SAL) and a case study application of SAL to the design of decentralized controllers for thermal regulation.

• The Spatial Aggregation Language

This thesis details an implemented programming language [6] for modeling and
controlling distributed physical systems. The Spatial Aggregation Language operators abstract common computational patterns in a wide class of data interpretation and control applications, and the accompanying programming environment supports rapid prototyping of such applications. The language explicitly encodes physical knowledge, allowing a variety of inference, explanation, tutoring, and design tasks.

- **Language definition**: This thesis defines a set of language data types and operators for representing and manipulating geometric and topological structures in distributed physical data at multiple levels of abstraction. The primitives in the language are geometric and topological representations of spatial objects that arise in distributed physical systems such as sound, fluid, and thermal fields. The compounds in the language are structured collections of spatial objects, such as spaces, neighborhood graphs, and equivalence classes. The abstraction mechanisms in the language convert compounds of spatial objects into single higher-level objects. The language operators construct and manipulate these objects and compounds and form abstractions by making explicit use of domain-specific physical knowledge, such as metrics, adjacency relations, and equivalence predicates.

- **Language implementation**: This thesis outlines the implementation of the SAL data types and operators as a C++ library. The library provides a consistent set of interfaces so that data types can be mixed and matched in the construction of a particular application.
- **Programming environment:** This thesis describes an implemented interactive environment interfaced to the SAL library to support rapid prototyping of SAL programs and interactive exploration of the structures that arise in an application instance.

- **Programming style:** This thesis explores the style of programming supported by SAL for applications that uncover and exploit structures in physical data. Style guidelines describe how to identify, encode, and utilize relevant physical knowledge in terms of appropriate SAL data types and operators.

- **Case study in decentralized control**

  This thesis also presents as a case study novel approaches to decentralized control design, in the context of thermal regulation. This case study develops novel algorithms for control placement and parameter design for systems with large numbers of coupled variables [3, 4, 5]. The SAL approach leverages physical knowledge to develop control design algorithms that are more efficient than standard techniques, and produce designs explainable in terms of problem structures. This case study also demonstrates the utility of the Spatial Aggregation Language operators in supporting these computations in a vocabulary natural for the domain.

- **Modeling heat flow:** This thesis demonstrates that Spatial Aggregation Language data types and operators support modeling of heat flow in a vocabulary natural for the domain.
Reasoning about influences: This thesis develops the SAL-based influence graph mechanism that utilizes physical knowledge of locality, linear superposability, and continuity to manipulate structural descriptions of influences in thermal fields. The influence graph serves as the basis for control design algorithms.

Control placement design: This thesis presents novel influence graph-based techniques for designing control node placement. The algorithms use structural knowledge to decompose a domain into regions so that controls in separate regions are maximally decoupled.

Control parameter design: This thesis presents novel influence graph-based algorithms for decentralized optimization of control parameters. The optimization algorithms use structural knowledge to efficiently evaluate temperature fields and to explicitly trade off computation, communication, and control quality during optimization.

1.3 Significance

The broad goal of this thesis is to provide powerful, high-level tools for distributed data analysis and control applications. The contributions described above support this goal and impact a wide variety of such applications.

The Spatial Aggregation Language is appropriate for applications which can leverage physical knowledge of locality and continuity to uncover and exploit multi-level structures in physical data (Section 3.5 discusses this in more detail.) For example, Spatial Aggregation has been applied to describing the common computational
requirements of problem solvers in application areas including dynamical system analysis [83, 88], nonlinear control [89, 90], and mechanical mechanism analysis [41]. Application areas currently being studied include weather data interpretation [38] and the analysis of diffusion-reaction morphogenesis [59].

The case study in decentralized control design introduces a new mechanism for reasoning about influences in physical fields. While the case study is centered on thermal regulation, the influence graph mechanism provides a generic vocabulary for designing decentralized controls, in terms of effects of controls on a field and similarities in control effects. These techniques are appropriate for other control design problems requiring placement and parametric optimization of decentralized controls for distributed physical fields.

1.4 Outline

The remainder of this thesis proceeds as follows. Chapter 2 discusses other work related to the Spatial Aggregation Language and decentralized control design. Chapter 3 presents the details of the language, focused on the core task of interpretation of spatially distributed physical data. Chapter 4 then applies and extends the language for modeling and control of distributed physical systems. Finally, Chapter 5 concludes and discusses future work.
CHAPTER 2

RELATED WORK

The work described in this thesis leverages results from a variety of fields, including reasoning about physical systems, structure discovery, spatial reasoning, engineering and scientific computing, and advanced programming languages. This chapter briefly reviews some of this related work and its connections to and differences from the Spatial Aggregation Language.

2.1 Reasoning about Physical Systems

Spatial Aggregation uses explicit representations of physical knowledge in order to reason about physical systems. Much research in Qualitative Reasoning has also studied how to apply high-level representations of physical systems and domain knowledge in order to predict, diagnose, reconfigure, and tutor [27, 18, 46, 80]. Many of the results in that community have centered around three main ontologies: the device ontology [18], which propagates qualitative constraints along topological connections between devices such as the elements of a circuit; the process ontology [27], which generates possible qualitative temporal evolutions of processes such as fluid flow between containers, based on specifications of interaction; and the constraint ontology [46],
which simulates qualitative differential equations describing the evolution of a system. High-level languages such as the Compositional Modeling Language [23] support the specification and compilation of domain knowledge. These ontologies tend to deal only with a system's topology, abstracting away its spatial properties.

Some recent research has extended these approaches to utilize spatial information. Qualitative spatial reasoning systems use abstract descriptions of shape and topology as the basis for inferring behaviors of systems. For example, the Region-Connection Calculus [16] represents topological relations, such as overlaps and is-disconnected-from, while Rajagopalan's extremal point representation [63] supports relative orientation and position descriptions. Qualitative physical fields [52] extend Qualitative Process Theory [27] to include qualitative spatio-temporal processes: for example, modeling heat flow between topologically connected sunny and shaded regions and inferring the evolution of warm and cold regions. Recognizing that topology is often not sufficient for complex tasks, the Metric Diagram / Place Vocabulary (MD/PV) theory [28] incorporates problem-specific metric information between special entities (places) in a domain. Similarly, the Spatial Semantic Hierarchy [48, 47] discovers "interesting" locations in the construction of mappings between topological and metric maps for robot navigation.

Spatial simulation research in the diagrammatic reasoning community also leverages knowledge of physical systems in order to predict behaviors over time. WHISPER [29] represents objects in a pixel-occupancy array in order to solve problems in a blocks world; similar models have been used to simulate fluid flow from low-level "molecular" interactions [31]. The analogical simulation framework [14] employs a
multi-level symbolic/visual representation of a system and has been used to simulate rigid body kinematic behaviors.

Spatial Aggregation differs from most of the above related work in that, in addition to using structural descriptions of physical phenomena to reason about systems, Spatial Aggregation also supports generation of the structural descriptions of physical phenomena from data or simulations ("predicate extraction" in the taxonomy of Chandrasekaran [13]).

The approach taken by Spatial Aggregation stems from work by Abelson et al. [1] underlining the importance of incorporating intelligence into scientific computing. Intelligent simulation uses a mixture of symbolic, numeric, and geometric techniques to automatically prepare, perform, and interpret simulations. Yip et al. [87] further refined this idea to specify a class of imagistic reasoning systems, in which the input is an image-like representation of spatially distributed physical data, the output is a high-level description of the data, and perception-like routines are used to perform the mapping. Yip and Zhao [86] then introduced Spatial Aggregation (SA) as a realization of imagistic reasoning unifying the mechanisms underlying a number of successful imagistic problem solvers, including KAM [83], which interprets the behaviors of Hamiltonian systems, MAPS [88], which designs control laws based on a geometric analysis of the state equations of a dynamical system, and HIPAIR [41], which analyzes the kinematics of fixed-axis mechanisms.

2.2 Structure Discovery

Spatial Aggregation applications seek to uncover and exploit structures that are implicitly represented in spatially distributed data. That goal is shared with a large
body of research from several related fields, including computer vision, scientific visualization, and data mining. The Spatial Aggregation Language builds on and generalizes techniques developed in these areas and incorporates them in a specific, efficient, high-level framework appropriate for a variety of distributed data interpretation and control tasks.

Image understanding systems generate high-level descriptions of scenes from input pixel arrays. The descriptions are often formed in terms of multi-level structures, such as segments and regions, uncovered by algorithms for edge detection and region growing and splitting [53]. Researchers have identified a number of primitive routines, such as boundary tracing and area activation [76], from which more complex visual processes can be composed.

Scientific visualization studies how physical data can be displayed in a manner that helps scientists use perceptual processes to understand it [65]. For example, weather data can be visualized using pseudocolor to represent temperature, iso-contours to connect points of equal pressure, needle diagrams to indicate directions of wind flow with arrows, streamlines to show connected flows, and animations of these to show changes over time. Interactive visualizations allow scientists to explore, focus, filter, project, and transform large data sets.

An important component of scientific visualization is the detection of features in data. For example, interpretation of fluid data simulations requires identification of vortices — regions of similarly swirling fluid. Visualization algorithms such as marching cubes [50] are often used to efficiently find the iso-surfaces that delineate such uniform regions. Advanced data interpretation techniques, such as the visiometrics algorithms [67] and the Fluid Dynamicist’s Workbench [84] can then be layered
over feature detectors in order to track structures over time, detecting births, deaths, splits, and merges. Alternatively, structures detected at one time instant can be modified to be consistent with the features at the next time instant, eliminating the need for repeated bottom-up recomputation of structures [59]. History-based algorithms have also been applied to the interpretation of seismic data, tracking waves as they propagate and reflect [42].

Data mining algorithms apply artificial intelligence and statistical techniques to search for patterns and correlations in databases [25]. In particular, spatial data mining algorithms identify correlations in spatial data. For example, mining of correspondences in geographic and weather data can detect general climate patterns across regions [51]. Important tasks in spatial data mining include clustering, generalizing, and classifying. Clustering algorithms partition a set of objects into groups of similar objects, using efficient specializations of statistical techniques. For example, CLARANS [58] incrementally improves a clustering by searching through similar randomly-generated changes; DBSCAN [22] uses information about cluster densities to reduce the effects of noise; DBCLASD [82] uses nearest-neighbor comparisons to find arbitrarily-shaped clusters in point data; STING [78] uses statistics at multiple spatial scales to efficiently search for commonalities. Many of these techniques avoid batch processing of all data by applying spatial indices to retrieve only relevant information at a given time. The Spatial Aggregation approach to clustering is a particularly efficient algorithm that leverages locality and continuity to find groups of similar objects. Spatial Aggregation also performs clustering at multiple levels of abstraction, using a common vocabulary instantiated with task- and level-specific knowledge.
Spatial data mining seeks generalizations — correlations, patterns, and trends — in massive spatial data sets. Some algorithms examine clusters for commonalities, using hierarchies in spatial features (e.g. cities in states in regions in countries) and nonspatial features (e.g. crop types) in order to move from specific examples to generalizations [45]. Other algorithms apply machine learning techniques to automatically uncover such trends; for example, by noticing common features of example clusters labeled as being stars in a database of sky images [24]. Generalizations serve as the basis for inducing classification rules (e.g. decision trees [62]) to apply to new data sets. These techniques have proved quite useful in analyzing massive scientific data sets; for example, in automatically cataloging sky images [24], identifying volcanos in images of the surface of Venus [11], and tracking cyclones in weather data [75]. It remains interesting future research to explore expressions of such generalization and classification algorithms in the Spatial Aggregation Language.

2.3 Spatial Reasoning

Applications such as geographic information systems (GIS), computer-aided design (CAD) systems, and robotics require reasoning about representations of spatial data. For example, a GIS user might want to correlate information in a collection of maps containing different data (e.g. a road map, a utilities map, and a forestry map); a CAD designer might wish to determine interaction of parts in a design; a robotics application might need to plan paths through an environment. These tasks are supported by techniques from computer graphics, computational geometry, and spatial databases for representing, manipulating, and querying spatial data. The Spatial
Aggregation Language makes use of similar techniques appropriate for its application domains.

Of particular relevance to the design of the Spatial Aggregation Language are the various representations of spatial objects [26]. At the lowest level, SAL utilizes a data-massive spatially distributed format for its data, as do many field-based geographic information systems and pixel-/voxel-based computer graphics algorithms. For the structures it uncovers in the input data, SAL utilizes more abstract representations. Parametric representations, such as splines, are useful for descriptions of smooth objects. Topological boundary representations, on the other hand, are useful for descriptions of objects formed of discrete components, such as a cube composed of square faces composed of perpendicular segments.

The Spatial Aggregation Language also makes use of the work in computational geometry and spatial databases on efficient indices for querying spatial data [66]. For example, $k$-d trees support efficient point location queries, while Voronoi diagrams support efficient nearest-neighbor location queries. SAL also utilizes computational geometry techniques for building adjacency graphs such as Delaunay triangulations, relative neighborhood graphs, and minimal spanning trees [17].

2.4 Engineering and Scientific Computing

A large body of engineering literature explores methods for modeling spatially distributed physical systems and elaborating the consequences of these models. Since
closed-form analytical solutions are often impossible, engineers typically use techniques such as finite differences and finite elements [91] to represent a system's governing partial differential equations in terms of matrices on an appropriate discretization. They then apply iterative algorithms [61] to solve the resulting sets of equations. Advanced techniques such as domain decomposition [12] and multigrid methods [10] achieve additional efficiency in convergence or parallelizability of computation.

The Spatial Aggregation Language differs from these traditional techniques in a number of ways. SAL provides operators and data types at a level of abstraction appropriate for the tasks, not requiring coercion of a program into a matrix form. It builds only local models and elaborates the consequences of these models through local interaction rules. It makes explicit where and how physical knowledge and domain-specific assumptions are being used, in order to avoid the fragility often associated with numerical methods. In combination with multi-layer descriptions of a system, the explicit use of physical knowledge allows SAL applications to explain results.

Advanced programming environments such as Matlab™ and Mathematica™ support construction of science and engineering applications by providing interpreted, high-level languages and interactive graphical environments. The Spatial Aggregation Language seeks to extend these ideas to naturally interface with spatially distributed application domains.

Much engineering research has also studied the design of decentralized control actions for spatially distributed phenomena. One approach is to simplify (e.g. linearize) the model of a system and apply traditional engineering techniques (e.g. linear-quadratic-gaussian control or Kalman filters) to the design [68]. Another approach is to apply local control methods at the individual controllers and then use hierarchical
techniques to exchange information necessary for global control [37]. Market-based methods [81] allow individual controllers to negotiate commodities representing control parameters in order to reach a global solution [32].

Design techniques for decentralized control placement have also been studied in the engineering community. Different metrics can be used to estimate the quality of a control design. For example, a control design can be evaluated in terms of effectiveness for specific vibration modes [30], required control energy [20], or error with respect to a desired control profile over a family of expected disturbances [34]. The controller placement is then computed by combinatorial optimization of the metric; for example, by greedy search [73, 33], genetic algorithms [20, 30, 49], or simulated annealing [15].

In contrast to these parametric and structural design techniques, the Spatial Aggregation Language seeks to use domain knowledge to extract and exploit high-level structural descriptions of physical phenomena in the design process. This yields principled methods for reasoning about designs and design trade-offs, based on an encapsulation of deep knowledge in structures uncovered for a particular problem. This in turn supports higher-level reasoning about and explanation of the design decisions.

2.5 Advanced Programming Languages

As a language for programming data interpretation and control tasks for distributed physical systems, the Spatial Aggregation Language provides not only suitable topological and geometric data types and operators necessary for such applications, but also a framework that embodies an appropriate style of programming.
In particular, it shares two features common to more general-purpose advanced programming languages: it is collection-oriented, and objects are related through local constraint propagation rules.

Collection-oriented languages [72] provide operations that act on collections of objects as a whole. For example, collection-oriented operations select members of, permute, form the union of, and perform operations on each element of a set. These techniques have proved useful in a number of distributed simulation environments. For example, SWARM [56] programs specify recursive collections of distributed agents, called swarms, which share behavioral code and a task agenda mechanism. Similarly, StarLogo [64] programs are constructed by specifying programs to be run in parallel on a number of distributed "turtle" agents that move around over stationary "patch" agents. These programming languages have been used for a number of distributed simulation applications, ranging from environmental to traffic simulation. The collection-oriented operations make it easy to program these applications at a high level of abstraction.

Constraint languages [74] allow objects to interact with each other by propagating information derived from local constraint rules. For example, specification of the voltage at one end of a resistor in an electrical circuit and the current through the resistor allows the inference of the voltage at the other end of the resistor. The Spatial Aggregation Language supports similar techniques in a spatially distributed context, for example, updating the temperature at a node based on heat propagation rules and the temperatures at surrounding nodes.
CHAPTER 3

THE SPATIAL AGGREGATION LANGUAGE

This chapter describes the design and implementation of the Spatial Aggregation Language (SAL) for modeling and controlling distributed physical systems. This chapter establishes the key, generic parts of the language, useful for a variety of distributed data interpretation, modeling, and control tasks. The next chapter specializes and extends the language with specific data types and operators supporting modeling and decentralized control design for distributed physical systems.

This chapter proceeds as follows. First, Section 3.1 introduces the underlying Spatial Aggregation framework. Then Section 3.2 specifies the language in terms of supporting data types and operators, and Section 3.3 describes the language implementation as a C++ library and an interactive, interpreted programming environment for rapid prototyping of SAL-based applications. Section 3.4 demonstrates the language style through several example programs, and Section 3.5 discusses the programming style common to these and other SAL-based applications.
3.1 Spatial Aggregation Framework

The Spatial Aggregation framework, developed by Yip and Zhao [86], supports specification of interpretation and control tasks for spatially distributed physical systems. It provides a small number of operators, parameterized by domain-specific knowledge, which can be applied to a set of uniform data types. This section briefly reviews Spatial Aggregation in order to lay the groundwork for its implementation in the Spatial Aggregation Language. Four key ideas of the Spatial Aggregation framework are particularly important to the development of SAL:

- **Field ontology**: Spatial Aggregation organizes computation around image-like representations of spatially distributed input data.

- **Multi-layer spatial aggregates**: Structures are uncovered at multiple levels of abstraction, with the structures uncovered at one level becoming the input to the structure-discovery process at the next level.

- **Uniform vocabulary**: Spatial Aggregation provides a small set of uniform data types and concise operators for navigating the spatial aggregate hierarchy. The data types and operators make explicit use of domain-specific knowledge.

- **Structure-based control design**: Spatial Aggregation control design applications utilize high-level, structural interpretations as the basis for determining low-level control actions.

The following sections elaborate on these ideas.
3.1.1 Field Ontology

A common characteristic of Spatial Aggregation applications, such as fluid data interpretation [85] and phase space-based control [89], is that the task is described in terms of discovery and manipulation of structures in spatially distributed data. Spatial Aggregation supports this process through imagistic reasoning [87] techniques: the input is an image-like representation of the spatially-distributed data, the output is a structural description, and the input-output mapping is performed by vision-like operations. In particular, Spatial Aggregation introduces the field ontology, in which the input is a field mapping from one continuum to another. For example, a 2-D temperature field associates a temperature with each point, mapping from $\mathbb{R}^2$ to $\mathbb{R}^1$ (Figure 3.1(a)); a 2-D fluid field associates a velocity with each point, mapping from $\mathbb{R}^2$ to $\mathbb{R}^2$ (Figure 3.1(b)).

A field is information-rich, in that its representation requires many bits. The identification of structures in a field is a form of data reduction: the information-rich field representation is abstracted into a more concise structural representation. For
example, a set of points on a curve can be described more compactly by a parameterized spline — the spline parameters are a much more concise representation than the enumeration of points.

3.1.2 Multi-Level Spatial Aggregates

Structural representations of fields exist at multiple scales and multiple levels of abstraction. For example, computer vision systems often extract segments, regions, and bodies from an input pixel field. Similarly, Spatial Aggregation supports multi-level structural descriptions of physical fields. For example, in a weather data analysis application [38], Spatial Aggregation could extract from pressure data the isobars, pressure cells, and pressure troughs.

Multi-layer structures arise from continuities in fields at multiple scales. Due to continuity, fields exhibit regions of uniformity, and these regions of uniformity can be abstracted as higher-level structures which in turn exhibit their own continuities. Task-specific domain knowledge specifies metrics and defines similarity and closeness of both field objects and their features. For example, isothermal contours are connected curves of equal (or similar enough) temperature.

Navigating the mapping from field to abstract description through multiple layers rather than in one giant step allows the construction of more modular programs with more manageable pieces that can use similar processing techniques at different levels of abstraction. The multi-level mapping also allows higher-level layers to use global properties of lower-level objects as local properties of the higher-level objects. For example, the average temperature in a region is a global property when considered
with respect to the temperature data points, but a local property when considered
with respect to a more abstract region description.

3.1.3 Uniform Vocabulary

Spatial Aggregation supports structure discovery through a small set of generic
operators, parameterized with domain-specific knowledge, on a uniform data type
(see Figure 3.2). Yip and Zhao [86] present a number of different problem solvers,
ranging from dynamical systems analysis to mechanical mechanism analysis, in terms
of the same set of generic operators parameterized by different domain knowledge.

The uniform data type of Spatial Aggregation, the *neighborhood graph*, is an ex-

cplicit representation of an object adjacency relation. The definition of adjacency
is domain-specific and depends on the metric properties of the input field. Com-
mon adjacency relations include Delaunay triangulations, minimal spanning trees,
and uniform grids. The neighborhood graph serves as computational glue, localizing
interactions between neighboring objects.

A particularly important use of neighborhood graphs in Spatial Aggregation is
as part of an efficient mechanism for uncovering structures. Higher-level structures
are extracted as *equivalence classes* of connected components within a neighborhood
graph relative to some domain-specific predicate. For example, isothermal regions in
a temperature field are connected points whose temperatures fall into the same bin.

The main Spatial Aggregation operators to manipulate objects and neighborhood
graphs include the following:

- **aggregate**: Builds a neighborhood graph from a specific adjacency specification.
Figure 3.2: Spatial Aggregation multi-layer spatial aggregates, uncovered by a uniform vocabulary of operators utilizing domain knowledge.
• **classify**: Finds equivalence classes of neighboring nodes in a neighborhood graph, using a specific equivalence predicate.

• **redescribe**: Transforms equivalence classes into higher-level objects.

• **localize**: Transforms higher-level objects back into their constituent equivalence classes.

Additional operators search through neighborhood graphs, check consistency of objects, extract geometric properties, and so forth.

By instantiating these operators with proper knowledge at different levels of abstraction, Spatial Aggregation allows specification of a variety of problem solvers.

### 3.1.4 Structure-Based Control Design

In addition to helping understand and explain physical data, the structures uncovered in physical data by Spatial Aggregation are useful in control design. For example, a dynamical system can be represented by a phase space, representing components of a system's state (both position and velocity) along different dimensions. The behaviors of the system can then be abstracted in terms of "flow pipes" of trajectories through the system's phase space, and a control task can be cast as a search through the flow pipes from the current state to some desired goal state [88]. Similarly, the behaviors of a mechanical mechanism can be described in terms of the mechanism's configuration space, with one dimension for each degree of freedom. Controlling the mechanism requires moving its state through connected physically-possible regions in the configuration space. Finally, this thesis presents novel methods that use structural representations of thermal fields in order to design decentralized controls for temperature regulation.
3.1.5 Simple Example: Trajectory Bundling

As an example of applying Spatial Aggregation to specify an imagistic reasoning application, consider a simple trajectory bundler, drastically simplified from Yip’s KAM program for analysis of dynamical systems [83]. In this application, the input is a set of points representing states of a dynamical system (i.e. points in the system’s phase space). Figure 3.4(a) shows example input points. Over time, the system’s state evolves, defining a mapping from one point to the next. The goal of the trajectory bundler is to find states that, in the limit, have the same behavior. This is done by uncovering structures at two levels of abstraction: trajectory curves of points in a sequence, and trajectory bundles of trajectory curves with similar shapes.

The trajectory bundling application can be specified in Spatial Aggregation by the following steps (see Figure 3.3):

1. Points to trajectory curves

   (a) Given input points (Figure 3.4(a)).

   (b) Aggregate the points into a minimal spanning tree (Figure 3.4(b)).

   (c) Classify connected points into the same equivalence class if the edge connecting them isn’t too long relative to nearby edges (Figure 3.4(c)).

   (d) Redescribe equivalence classes of consistent points as trajectory curves (Figure 3.4(d)).

2. Trajectory curves to trajectory bundles
(a) Aggregate trajectory curves such that curves are adjacent if any of their constituent points are neighbors in the underlying minimal spanning tree (Figure 3.4(e)).

(b) Classify connected curves into the same equivalence class if their shape is relatively similar (Figure 3.4(f)).

(c) Redescribe equivalence classes of consistent trajectory curves as trajectory bundles.

This example demonstrates the common computational structure between the abstraction levels and the incorporation of domain-specific knowledge of locality and similarity in the Spatial Aggregation operations. Sections 3.4.1 and 3.4.2 explore the SAL implementation of this application.

3.2 Spatial Aggregation Language Definition

The Spatial Aggregation Language provides data types and operators supporting development of data interpretation, modeling, and control applications for distributed physical systems. Following the Spatial Aggregation framework presented in the previous section, SAL components make explicit use of physical knowledge to uncover multi-layer aggregates of structures in physical fields. Table 3.1 categorizes the components into primitives, compounds, and means of abstraction, similarly to components in general-purpose programming languages [2].

The following subsections describe these components in more detail. Each subsection provides a high-level overview of a component: its significance for distributed data interpretation, modeling, and control applications, the domain-specific physical knowledge it exploits, and the syntax for some of the main operations associated with
Figure 3.3: Spatial Aggregation data flow for trajectory bundling application.
Figure 3.4: Example steps in trajectory bundling application. (a) Input points. (b) Points aggregated into a minimal spanning tree. (c) Equivalence classes of points joined by short-enough edges. (d) Equivalence classes redescribed as trajectories. (e) Trajectories aggregated based on adjacencies of constituent points. (f) Equivalence classes of trajectories with similar-enough limit behavior.
• **Primitive Objects** represent structures in spatial data.
  Example: 

• **Compound Objects** combine primitive objects.
  - *Spaces* group objects.
    Example (points and curves):
  - *Fields* associate objects and features.
    Example (points and temperatures):
  - *Ngraphs* relate nearby objects.
    Example (Delaunay triangulation):
  - *Equivalence classes* group similar objects.
    Example (points with similar vector directions):

• **Means of Abstraction** connect compound objects at one level of abstraction and primitive objects at the next.
  Example (points to region bounded by convex hull):

Table 3.1: Components of the Spatial Aggregation Language.
the component. The programming language is supported by a large library of data
type implementations, discussed in the next section and detailed in an appendix: ad-
ditional high-level operations are also layered over the main operations listed here.
The space and time complexity of the various operators depends on the implemen-
tations chosen. For example, a 2-d Delaunay triangulation can be computed in time
$O(n \log n)$, while a fully-connected neighborhood graph requires time $O(n^2)$. The
appendix provides more information about complexity where appropriate.

3.2.1 Spatial Object

The primitive objects in a SAL application are structures in spatially-distributed
data at multiple levels of abstraction. SAL provides the spatial object data type to
represent these objects.

Definition 1 (Spatial Object) A spatial object represents the structure and extent
of a portion of space.

Figure 3.5 illustrates some spatial objects in a meteorology application, where
points denote sampled data, curves represent isobars connecting points of equal pres-
sure, and regions indicate low pressure cells [38].

In Spatial Aggregation, it is important to distinguish between the topological struc-
ture of a spatial object and the geometric properties (e.g. edge length, angle, curvature,
area) of that structure with respect to a particular reference frame. For example, the
structure of a quadrilateral can be specified simply by providing a sequence of contigu-
ous edges. By specifying different coordinates for the vertices of the object, different
Figure 3.5: Some spatial objects in a meteorology application: sample points, isobar curves, and a region of low pressure.

geometric shapes can be defined. The resulting geometric objects can have similar properties (compare Figure 3.6(a) and Figure 3.6(b)) or very different properties (compare Figure 3.6(a) and Figure 3.6(c)).

This distinction can be important in several places in SAL programs. In some cases only the structure is important — there is no need to compute the geometric properties. In other cases, the natural computational flow allows construction of structural connections before the information necessary for geometric properties is available. Often the same structure can be viewed with respect to different reference frames (e.g. a local coordinate system vs. a global coordinate system, or a projection into fewer dimensions), so different geometric properties are computed for the same structure.
Figure 3.6: The same topological structure with different coordinates.

Topological Structure

The topological structure of an object specifies how its parts are related to each other. In some cases, the structure is defined implicitly along with the geometry; for example, consider the disk defined by $x^2 + y^2 \leq r^2$. In other cases, however, the structure is explicitly constructed: for example, by specifying the faces of a cube (space/subspace relation) or by specifying a triangulation of a polygon (adjacency relation).

In order to capture the common representational requirements of programs uncovering multi-layer structures in spatially distributed physical data, the Spatial Aggregation Language supports explicit structure construction with the cell complex representation [57]. Cell complexes provide a powerful, generic mechanism for building hierarchical representations of the structures in physical fields in terms of primitive cells.
Definition 2 (Cell) A k-cell is a region of space that is homeomorphic (i.e. can be continuously deformed, without tearing) to a k-dimensional ball.

Examples of cells include a point (0-cell); a closed line segment or a closed curve segment (1-cell); a triangle with its interior or a surface patch (2-cell); and a solid cube (3-cell). Each of these can be deformed to a ball of the same dimension.

Cells have hierarchical structure: cells at one dimension are the proper faces of cells at the next higher dimension. For example, points (dimension 0) are the proper faces of line segments (dimension 1), which are the proper faces of polygons (dimension 2), which are the proper faces of polyhedra (dimension 3), and so forth. Figure 3.7 shows some examples of this hierarchy. An object's faces include its proper faces, their proper faces, and so forth.

More complex topological structures are built by combining cells in a disciplined manner.

Definition 3 (Cell Complex) A cell complex \( X = \{c_1, c_2, \ldots, c_n\} \) is a collection of cells obeying the following properties:

1. Each cell's faces are in the complex.

2. A non-empty intersection of two cells is a face of each.

Figure 3.8 shows example legal cell complexes, and Figure 3.9 shows example illegal complexes that violate the intersection property.

An atomic cell complex is the complex consisting of a cell and its faces, while a non-atomic cell complex is a complex for multiple cells. For example, in Figure 3.10(a), the set consisting of the triangle, its edges, and its vertices is an atomic cell complex, and
Figure 3.7: Example cells. Cells of dimension $n$ are the proper faces of cells of dimension $n + 1$.

Figure 3.8: Legal cell complexes.

Figure 3.9: Illegal cell complexes.
in Figure 3.10(b), the set consisting of the four triangles, their edges, and their vertices is a non-atomic cell complex. Note that the non-atomic complex is not equivalent to the corresponding atomic cell complex defined by a hexagon, its edges, and its vertices. In some cases (e.g. to support efficient querying) it is advantageous to associate with an atomic complex the corresponding non-atomic triangulation complex.

Cell complexes can answer queries about their faces and the relations among them. There are three types of relations: face relation, co-face relation, and same-dimension adjacency of faces. The face and co-face relation specify the hierarchical structure of cells in the complex: the face relation bottom-up (e.g. points are faces of segments which are faces of polygons) and the co-face relation top-down (e.g. polygons are co-faces of segments which are co-faces of points). The adjacency relation is established if two objects of the same dimension share a face (e.g. two contiguous line segments).

As an example of these relations, consider the complex in Figure 3.11. Relations are shown with dotted lines (only the proper relations are shown): face relations are marked "fc." co-face relations are marked "cf," and adjacency relations are marked
Figure 3.11: The structural relations of a quadrilateral cell complex. Dotted lines denote proper face, co-face, and adjacency relations.

"adj." For example, one face of the quadrilateral $ABCD$ is the line segment $AB$; one face of that line segment is the point $A$. In the opposite direction, one co-face of point $A$ is line segment $AB$; one co-face of that line segment is the quadrilateral. Finally, line segments $AB$ and $BC$ are adjacent since they share point $B$.

Cell complexes can be constructed by combining cells, or as instances of prepackaged specific implementations, such as cell complexes for segments, and polygons. These specific types of cell complexes support additional functionality particular to
• **dimension**: $(\text{Cell}_\text{Complex} \ x) \rightarrow (\text{Number} \ d)$
  - Returns the highest-dimension cell in the complex.
  - $d = \max \{k \in \mathcal{V} : (\exists c \in x : c \text{ is a } k - \text{cell})\}$.

• **is_adjacent**: $(\text{Cell}_\text{Complex} \ x, \text{Cell} \ c_1, \text{Cell} \ c_2) \rightarrow (\text{Boolean} \ b)$
  - Returns whether or not $c_2$ is adjacent to $c_1$ in the complex.
  - $b = (c_1 \in x) \land (c_2 \in x) \land (c_1 \neq c_2) \land (c_1 \cap c_2 \neq \emptyset) \land (\exists k : c_1 \text{ and } c_2 \text{ are } k - \text{cells})$.

• **is_coface**: $(\text{Cell}_\text{Complex} \ x, \text{Cell} \ c_1, \text{Cell} \ c_2) \rightarrow (\text{Boolean} \ b)$
  - Returns whether or not $c_2$ is a co-face of $c_1$ in the complex.
  - $b = (c_1 \in x) \land (c_2 \in x) \land (c_1 \neq c_2) \land (c_1 \cap c_2 = c_1)$. 

• **is_face**: $(\text{Cell}_\text{Complex} \ x, \text{Cell} \ c_1, \text{Cell} \ c_2) \rightarrow (\text{Boolean} \ b)$
  - Returns whether or not $c_2$ is a face of $c_1$ in the complex.
  - $b = (c_1 \in x) \land (c_2 \in x) \land (c_1 \neq c_2) \land (c_1 \cap c_2 = c_2)$.

Table 3.2: Major cell complex operations.

The geometric properties for a spatial object depend on the particular type of object; for example, a point has coordinates, a segment has length, a curve has curvature at a specified point, and a polygon has angles. Given a topological structure...
and a reference frame, a geometric object caches the appropriate geometric properties for the structure with respect to the reference frame.

A metric space defines the distance function (metric) and reference frame necessary to define geometric properties. A common example of a metric space is the two-dimensional Euclidean space, where the metric is defined by the familiar distance function for points \( p \) and \( q \):

\[
d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.
\]

**Definition 4 (Metric Space)** A metric space for a collection of objects \( S \) defines a metric function \( d : S \times S \rightarrow \mathbb{R} \) satisfying the following properties, for \( x, y, z \in S \):

1. \( d(x, y) \geq 0 \).
2. \( d(x, y) = 0 \) iff \( x = y \).
3. \( d(x, y) = d(y, x) \).
4. \( d(x, z) \leq d(x, y) + d(y, z) \).

In fact, the Euclidean metric space is a particular kind of metric space, a coordinate space, which assigns points coordinates in some coordinate system and measures distances between points with respect to their coordinates. The 2-D Euclidean coordinate space \( \mathbb{R}^2 \) uses \( (x, y) \) coordinate pairs and measures distances with the Euclidean metric specified above, while the 2-D Manhattan coordinate space uses similar coordinate pairs but a different metric, the difference in \( x \) values plus the difference in \( y \) values, as if measuring city blocks. Finally, a 2-D polar coordinate space uses \( (r, \theta) \) coordinate pairs and yet another formula for measuring distances.

Metric spaces for complex geometric objects can use structure in addition to point coordinates. For example, a metric space for polygons might assign a unique point
(say the centroid) to each object and use a coordinate space on the points to measure distances.

Given a topological structure and a metric space, geometric properties can be computed. This can be done as part of the algebraic definition of the structure (e.g. $x^2 + y^2 \leq r^2$), or by using geometric properties of other parts of the structure. Bottom-up geometric property calculation uses geometric information of a structure's faces to determine properties of the structure itself; for example, computing a triangle's area and angles based on the lengths of its sides. Alternatively, top-down geometric property calculation establishes geometric properties for a structure's faces based on the geometry of the structure; for example, using the angles and area of a triangle to define the lengths of its sides. Similarly to cell complexes, SAL lets the user explicitly create geometric objects and define their properties, or choose from a variety of useful prepackaged objects defined in the library.

The metric space defining the geometric properties of a structure is known as the embedding space: for example, a square can be embedded in a 3-D space, as shown in Figure 3.12. The geometric object in turn then defines a local metric space for its substructure: for example, a curve defines a 1-D parameterization and a square defines a 2-D parameterization, as shown in Figure 3.13. Note that this entails two separate sets of geometric properties for an object's substructure — with respect to the structure's embedding space and with respect to its local space — underscoring the need for separation of structure definition and geometry definition.

Geometric properties computed for a structure in one metric space can be transformed to another, related metric space. A common transformation is to compute
geometry for an object's substructure with respect to the local space and then transform it to the object's embedding space, as shown in Figure 3.14(a), or vice-versa. Similarly, if two metric spaces are related by a known transformation (e.g. translation or rotation), geometric properties can be efficiently computed with respect to one space given properties with respect to the other. In Figure 3.14(b), the transformation between rotated Euclidean metric spaces changes coordinates but preserves angles and areas.

In summary, the SAL geometric object representation extends a spatial object with geometric properties relative to metric spaces.

**Definition 5** Geometric Object A geometric object $G$ is a tuple $(O, E, L, P)$ where
Figure 3.14: Geometry transformations. (a) Between local metric space and embedding metric space (point's coordinates on edge to/from point's coordinates in rectangle); (b) between transformed metric spaces (2-D Euclidean to/from rotated 2-D Euclidean).

- \( O \) is a spatial object for the structure.

- \( E \) is an embedding metric space.

- \( L \) is a local metric space.

- \( P \) is a set of object-specific geometric properties.

Table 3.3 lists major operations for geometric objects.

### 3.2.2 Compound Objects

Compound objects in the Spatial Aggregation Language are collections of primitive objects or other compounds. SAL defines a number of compound types with special semantics, including spaces, fields, neighborhood graphs, and equivalence classes.
- **distance**: \((\text{Geometric\_Object } o_1, \text{Geometric\_Object } o_2) \rightarrow \text{(Number } n)\)
  - Returns the distance between the two objects with respect to their common embedding metric space.
  - \( n = d[E[o_1]](o_1, o_2), \text{ when } E[o_1] = E[o_2]. \)

- **embedding\_space**: \((\text{Geometric\_Object } o) \rightarrow \text{(Metric\_Space } s)\)
  - Returns the space with respect to which the object is defined.
  - \( s = E[o]. \)

- **local\_space**: \((\text{Geometric\_Object } o) \rightarrow \text{(Metric\_Space } s)\)
  - Returns the space parameterizing the object.
  - \( s = L[o]. \)

- **structure**: \((\text{Geometric\_Object } o) \rightarrow \text{(Spatial\_Object } s)\)
  - Returns the structure for which \( o \) defines geometric properties.
  - \( s = O[o]. \)

- **volume**: \((\text{Geometric\_Object } o) \rightarrow \text{(Number } v)\)
  - Returns the generalized volume (length, area, etc.) for the object.
  - \( v = \text{volume property of } P[o]. \)

| Table 3.3: Major geometric object operations. |
Figure 3.15: A space collects spatial objects for element-wise and global operations.

Space

A space (Figure 3.15) is simply a collection of spatial objects, and serves as the basis for operating on a number of objects simultaneously, either independently or to extract a global property.

Definition 6 (Space) A space \( S \) is a set \( \{o_1, o_2, \ldots, o_n\} \), where each \( o_i \) is a spatial object.

Element-wise operations for a space are implicitly distributed over the objects in a space. For example, an operation to compute area could be passed out to all the polygon objects in some set. Global properties are extracted by combining results from members of the set. For example, the average area of the polygon collection could be computing by gathering from each polygon its area, summing, and dividing by the number of members. Spaces are an important component of SAL supporting high-level programming over collections of distributed data sources and control nodes.
The Spatial Aggregation Language provides collection-oriented operations manipulating spaces themselves. Common operations include selecting a subspace that satisfies a particular predicate (e.g. regions larger than a certain size), forming a related space (e.g. the space of points translated by 2 units from those in a given space), and finding the union, intersection, and difference of a pair of spaces.

Metric spaces were discussed earlier as a mechanism for establishing geometric properties for spatial objects. Metric spaces also serve as containers for the geometric objects embedded in them, and can implement spatial indices to efficiently answer distance-related queries such as “which object is nearest to that one?” and “what objects are within a distance of 5 of this one?” Common indices include grids, k-d trees, and Voronoi diagrams. This spatial indexing strategy is especially useful in conjunction with subspace selection. All geometric objects can be defined with respect to a common base metric space, and then a particular subset can be selected and indexed in a derived metric space. For example, the base metric space could include points, curves, and regions; a certain set of points could be selected and indexed in a derived k-d tree. The selection process does not require recomputation of geometric properties but makes available powerful query mechanisms suitable for the selected objects.

Table 3.4 lists major operators for manipulating distributed sets of objects with spaces.

Cell Complexes as Collections

Although cell complexes were defined as the representation for primitive spatial objects, a cell complex is in fact a structured collection of other primitive objects — its faces. In particular, cell complexes define face, co-face, and adjacency queries.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>gather</strong> : (Space s. Object i. (Object * Spatial_Object → Object) p) → (Object g)</td>
<td></td>
</tr>
<tr>
<td>- Starting with the initial object i, combines each object from the space to the current result.</td>
<td></td>
</tr>
<tr>
<td>- ( g = i \oplus o_1 \oplus o_2 \oplus \ldots \oplus o_n ) where ( \oplus ) represents function ( p ), and ( s = {o_1, o_2, \ldots, o_n} ).</td>
<td></td>
</tr>
<tr>
<td><strong>intersection</strong> : (Space s1, Space s2) → (Space i)</td>
<td></td>
</tr>
<tr>
<td>- Returns the intersection of the two spaces.</td>
<td></td>
</tr>
<tr>
<td>- ( i = s_1 \cap s_2 ).</td>
<td></td>
</tr>
<tr>
<td><strong>map</strong> : (Space s, (Spatial_Object → Spatial_Object) c) → (Space m)</td>
<td></td>
</tr>
<tr>
<td>- Returns a space with objects mapped from the members of ( s ) by the given function.</td>
<td></td>
</tr>
<tr>
<td>- ( m = {c(o) : o \in s} ).</td>
<td></td>
</tr>
<tr>
<td><strong>near</strong> : (Metric_Space s, Geometric_Object o, Number r) → (Space n)</td>
<td></td>
</tr>
<tr>
<td>- Returns a space of objects within distance ( d ) of ( o ).</td>
<td></td>
</tr>
<tr>
<td>- ( n = {o_2 \in s : d[s](o, o_2) \leq r} ).</td>
<td></td>
</tr>
<tr>
<td><strong>nearest</strong> : (Metric_Space s, Geometric_Object o) → (Geometric_Object r)</td>
<td></td>
</tr>
<tr>
<td>- Returns the member of ( s ) nearest to ( o ).</td>
<td></td>
</tr>
<tr>
<td>- ( n = \arg\min_{o_2 \in s} {d[s](o, o_2)} ).</td>
<td></td>
</tr>
<tr>
<td><strong>select</strong> : (Space s. (Spatial_Object → Boolean) t) → (Space l)</td>
<td></td>
</tr>
<tr>
<td>- Returns a space containing only those objects of ( s ) that pass the test.</td>
<td></td>
</tr>
<tr>
<td>- ( l = {o \in s : t(o)} ).</td>
<td></td>
</tr>
<tr>
<td><strong>union</strong> : (Space s1, Space s2) → (Space u)</td>
<td></td>
</tr>
<tr>
<td>- Returns the union of the two spaces.</td>
<td></td>
</tr>
<tr>
<td>- ( u = s_1 \cup s_2 ).</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: Major space operations.
Table 3.5: Collection-oriented cell complex operations.

that “open up” the complex and return, for a given member of the complex, a space containing the other members of the complex that satisfy the given relationship. For example, in Figure 3.11, a proper co-face query for vertex $A$ would return the space of segments $\{AB, AD\}$. Table 3.5 details these operations.

Field

The cornerstone of Spatial Aggregation is the field ontology: applications seek to find structures in image-like fields of spatially distributed data. Formally, a field is a mapping from one continuum (the domain space) to another (the feature space), for example $\mathbb{R}^n \rightarrow \mathbb{R}^m$. In actual programs, fields are usually discretized in both domain and feature spaces. A common discretization is in terms of (perhaps evenly-spaced) points; for example, an $\mathbb{R}^2 \rightarrow \mathbb{R}^1$ temperature field could be discretized as a set of discrete location points mapping to temperature values. Other discretizations are also possible; for example, a temperature field could be represented as a set of patches.
Figure 3.16: A field associates objects and features. (a) Association of points and temperature values. (b) Association of points and wind direction vectors.

with associated temperature intervals. The Spatial Aggregation Language represents fields as associations of objects in a domain metric space with objects in a feature metric space (Figure 3.16).

**Definition 7 (Field).** A field $F : D \rightarrow R$ is a mapping from objects in a domain metric space $D$ to objects in a feature metric space $R$.

Since the domain and feature spaces are discrete, a field can actually be implemented as a set of pairs; for example, the $R^3 \rightarrow R^1$ temperature field would be implemented as a collection of $(R^3$ point, $R^1$ temperature) pairs. This pair-based representation is appropriate for distributed data interpretation, modeling, and control applications, where a location naturally “owns” associated features (e.g. distributed sensors measuring temperature).

Similarly to spaces, fields support element-wise operations implicitly distributed among the members of the field. For example, to change units, a temperature field can be scaled and added to a constant field. To derive a squared-error field, the actual field can be subtracted from the desired field and the resulting field squared.
SAL also provides collection-oriented operations manipulating fields as units. Selection returns a subfield; this could be useful in finding the temperature field near a given sensor. Gathering supports collection of information about an entire field, such as the average, minimum, and maximum temperature. Finally, interpolation allows estimation of field values in discretized fields for objects not explicitly represented, for example, by considering nearby temperatures.

Table 3.6 and Table 3.7 list the major operators for manipulating distributed physical fields.

Ngraph

Spatially distributed systems exhibit locality: objects exist in regions of space and interact predominantly with nearby objects; global interactions are aggregated from collections of local interactions. Thus to program distributed data interpretation tasks, it is natural to work from local interpretations to more global ones. Similarly, to program distributed control tasks, it is natural to use local control actions to achieve global control objectives.

The Spatial Aggregation Language provides the neighborhood graph (ngraph) mechanism for defining task-specific locality (Figure 3.17). A neighborhood graph forms adjacencies among objects based on a specified neighborhood relation (e.g. minimal spanning tree or k-nearest neighbors). In particular, given a space of objects to aggregate, a neighborhood graph constructs a set of (object, neighbor) adjacency pairs.

**Definition 8 (Ngraph)** An ngraph $G = (S, A)$ where

- $S$ is a space.
- $A \subseteq S \times S$ is a set of adjacencies $(o_1, o_2)$ between objects in $S$. 

48
• **combine** : (Field f1, Field f2, (Geometric_Object * Geometric_Object → Geometric_Object) r) → (Field c)
  - Returns a field mapping objects in the union of domains of f1 and f2 to features defined by combining the corresponding features of f1 and f2 with the given function.
  - \( c : (D[f_1] \cup D[f_2]) \rightarrow (R[f_1] \cup R[f_2]) \) such that
    \[
    c(o) = \begin{cases} 
    r(f_1(o), f_2(o)) & \text{if } o \in D[f_1] \cap D[f_2] \\
    f_1(o) & \text{if } o \in D[f_1] - D[f_2] \\
    f_2(o) & \text{if } o \in D[f_2] - D[f_1]
    \end{cases}
    \]

• **domain_space** : (Field f) → (Space s)
  - Returns the domain space of the field.
  - \( s = D[f] \).

• **feature** : (Field f, Geometric_Object o) → (Geometric_Object r)
  - Returns the feature of the object from the field’s domain space.
  - \( r = f(o) \).

• **feature_space** : (Field f) → (Space s)
  - Returns the feature space of the field.
  - \( s = R[f] \).

• **gather** : (Field f, Object i, (Object * Geometric_Object * Geometric_Object → Object) p) → (Object g)
  - Starting with the initial object, combines each (domain, feature) pair from the field to the current result.
  - \( g = i \oplus (o_1, f(o_1)) \oplus (o_2, f(o_2)) \oplus \ldots \oplus (o_n, f(o_n)) \) where \( \oplus \) represents function p, and \( D[f] = \{o_1, o_2, \ldots, o_n\} \).

Table 3.6: Major field operations.
• **interpolate** : (Field f, Space d) → (Field i)

  - Returns a new field with features of additional domain objects defined by an implementation-specific interpolation function.

  - $i : (D[f] \cup d) \rightarrow (R[f] \cup R_2)$ such that

    $$i(o) = \begin{cases} f(o) & \text{if } o \in D[f] \\ \text{some } r \in R_2 & \text{else} \end{cases}$$

  where $R_2$ is implementation-specific.

• **map** : (Field f, (Geometric_Object → Geometric_Object) c) → (Field m)

  - Returns a field with the same domain as $f$ and features defined by applying the function to the features of $f$.

  - $m : D[f] \rightarrow R_2$ such that $m(o) = c(f(o))$ and $R_2 = \{c(r) : r \in R[f]\}$.

• **select** : (Field f, (Geometric_Object * Geometric_Object → Boolean) t) → (Field s)

  - Returns a field containing only those (domain, feature) pairs of $f$ that pass the test.

  - $s : D_2 \rightarrow R_2$ where $D_2 = \{o \in D[f] : t(o, f(o))\}$ and $R_2 = \{f(o) : o \in D_2\}$.

• **subfield** : (Field f, Space s) → (Field r)

  - Returns the restriction of $f$ to $s$.

  - $r : s \rightarrow R_2$ such that $r(o) = f(o)$ for $o \in s \cap R_2 = \{f(o) : o \in s\}$.

Table 3.7: Additional major field operations.
Figure 3.17: Ngraphs localize computation with object adjacencies. (a) Neighbors within a fixed radius. (b) Neighbors by Delaunay triangulation. (c) Neighbors in MST.

Note that while physical neighborhoods are symmetric, the definition of ngraph allows directed adjacencies. This can be useful computationally, allowing implementation of asymmetric neighborhood graphs, such as k-nearest neighbors. An adjacency in an undirected neighborhood graph has a corresponding back-adjacency.

Adjacencies in a neighborhood graph serve to localize computations, such that a node interacts only with its neighbors. Examples of local computations include comparisons (e.g. compare the wind direction at a node with that at its neighbors) and interaction rules (e.g. update the temperature at a node based on temperatures at surrounding nodes). Collection-oriented operations on neighborhood graphs include a variety of graph-theoretic operations, such as union, intersection, subgraph, and closure. These operations support distributed data interpretation, modeling, and control applications by providing a sophisticated vocabulary for building and manipulating local computation frameworks.

Specializations of neighborhood graph construct additional structure besides adjacencies. In particular, for some neighborhood graphs there exist cell complexes corresponding to the adjacency structure. For example, a minimal spanning tree
neighborhood graph has a corresponding complex consisting of segments connecting adjacent nodes. One particularly useful complex that can be built by a neighborhood graph is a mesh. A mesh partitions space into elements whose vertices are the nodes of the graph and whose edges correspond to the adjacencies of the graph. For example, consider the triangulation (a mesh whose elements are triangles) shown in Figure 3.18. Note that since the elements are themselves spatial objects, a dual neighborhood graph can be built by considering them to be nodes and forming adjacencies between elements that share edges.

Neighborhood graphs can also be specialized to take advantage of existing structure on the aggregated space. For example, suppose the nodes of a neighborhood graph have geometries in a rectilinear grid, as shown in Figure 3.19. A neighborhood graph can use this structure to support additional queries about neighbors in a particular direction.

Tables 3.8 and 3.9 list major operations for building and manipulating local computation frameworks with neighborhood graphs.
• **adjacencies**: \( (\text{Ngraph } g) \rightarrow (\text{Adjacencies } a) \)
  - Returns the adjacencies defined by \( g \).
  - \( a = A[g] \).

• **aggregate**: \( (\text{Space } s, \text{Ngraph Constructor } n) \rightarrow (\text{Ngraph } g) \)
  - Explicates the adjacency relation defined by \( n \) on \( s \).
  - \( g = (s,a) \) where the adjacencies \( a \) are implementation-specific according to \( n \).

• **cell_complex**: \( (\text{Ngraph } g) \rightarrow (\text{Cell_Complex } x) \)
  - Returns the cell complex corresponding to the neighborhood graph. Only valid for particular types of neighborhood graph.
  - \( x = S[g] \cup \{\text{segment}(o_1,o_2) \text{ for } (o_1,o_2) \in A[g]\} \) and implementation-specific higher-dimension cells.

• **closure**: \( (\text{Ngraph } g, \text{Number } n) \rightarrow (\text{Ngraph } c) \)
  - Returns an ngraph derived from \( g \) with an object’s neighbors being its neighbors’ neighbors’ \( \ldots \) (up to \( n \) times) in \( g \).
  - \( c = (S[g], a^n) \) where
    \[
    a^{i+1} = a^i \cup \{ (o_1,o_3) : (o_1,o_2) \in a^i \land (o_2,o_3) \in a^i \} \\
    a^0 = A[g]
    \]

• **directional_neighbor**: \( (\text{Grid } g, \text{Spatial_Object } o, \text{Number } a, \text{Number } d) \rightarrow (\text{Spatial_Object } n) \)
  - Returns the neighbor of \( o \) in \( g \) along the given axis \( a \) in either the positive or negative direction \( d \).
  - \( n \in S[g] \) such that \( (o,n) \in A[g] \land d * ((n - o) \cdot e_a) > 0 \) where \( e_a = (0,\ldots,1,\ldots,0) \) with 1 in the \( a \)th position.

• **intersection**: \( (\text{Ngraph } g1, \text{Ngraph } g2) \rightarrow (\text{Ngraph } i) \)
  - Returns an ngraph with adjacencies the intersection of the adjacencies in the two ngraphs.
  - \( i = (S[g_1] \cup S[g_2], A[g_1] \cap A[g_2]) \).

Table 3.8: Major neighborhood graph operations.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Input Types</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>neighbors</td>
<td>Ngraph g, Spatial_Object o</td>
<td>(Space n)</td>
</tr>
<tr>
<td></td>
<td>- Returns the neighbors of o in g.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- n = {o_2 \in S[g] : (o, o_2) \in A[g]}.</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>Ngraph g, (Adjacency \rightarrow Boolean) t</td>
<td>(Ngraph l)</td>
</tr>
<tr>
<td></td>
<td>- Returns an ngraph containing only those adjacencies of g that pass the test.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- l = (S[g], {a \in A[g] : t(a)}).</td>
<td></td>
</tr>
<tr>
<td>space</td>
<td>Ngraph g</td>
<td>(Space s)</td>
</tr>
<tr>
<td></td>
<td>- Returns the space aggregated by the ngraph.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- s = S[g].</td>
<td></td>
</tr>
<tr>
<td>subgraph</td>
<td>Ngraph g, Space s</td>
<td>(Ngraph b)</td>
</tr>
<tr>
<td></td>
<td>- Returns an ngraph derived from g with adjacencies only connecting the objects in the space.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- b = (s, {a \in A[g] : o_1[a] \in s \land o_2[a] \in s}).</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>Ngraph g_1, Ngraph g_2</td>
<td>(Ngraph u)</td>
</tr>
<tr>
<td></td>
<td>- Returns an ngraph with adjacencies the union of the adjacencies in the two ngraphs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- u = (S[g_1] \cup S[g_2], A[g_1] \cup A[g_2]).</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.9: Additional major neighborhood graph operations.
Figure 3.19: Rectilinear grid supports directional neighbor queries based on geometry of input space.

Equivalence Classes

The main goal in Spatial Aggregation is to find and use structures — regions of uniformity — in distributed physical data. The classifier mechanism in the Spatial Aggregation Language uses domain knowledge to find these equivalence classes. A classifier partitions a space into subspaces and can identify to which subspace each member belongs.

Definition 9 (Classifier) A classifier $C = (S, P)$ where

- $S$ is a space.

- $P \in 2^S$ partitions $S$ into a set of subsets $\{S_1, S_2, \ldots, S_n\}$.

Table 3.10 summarizes the operations available to query a classifier about the equivalence classes it defines.
- **classes**: 
  \((\text{Classifier } c) \rightarrow (\text{Spaces } p)\)
  
  - Returns the partitioning defined by the classifier.
  
  - \(p = P[c]\).

- **classify**: 
  \((\text{Space } s, \text{Classifier Constructor } n) \rightarrow (\text{Classifier } c)\)
  
  - Explicates the classification on \(s\).
  
  - \(c = (s, p)\) where the partition \(p\) is implementation-specific according to \(n\).

- **class.of**: 
  \((\text{Classifier } c, \text{Spatial Object } o) \rightarrow (\text{Space } s)\)
  
  - Returns the sub-space to which \(o\) belongs in the classification.
  
  - \(s \in P[c]\) such that \(o \in s\).

- **space**: 
  \((\text{Classifier } c) \rightarrow (\text{Space } s)\)
  
  - Returns the space partitioned by the classifier.
  
  - \(s = S[c]\).

Table 3.10: Major classifier operations.
Different classifier implementations use different processes to support the equivalence class partitioning. In particular, classifiers distinguish two key pieces of knowledge:

- **Clustering mechanism:** How to search for equivalence classes.
  Ex: transitive closure; merge/split algorithms.

- **Equivalence predicate:** When two objects belong to the same group.
  Ex: nearness in feature space; equality of feature bin.

One particularly efficient classification mechanism uses transitive closure of an equivalence predicate, with respect to a neighborhood graph. The mechanism proceeds as follows (illustrated in Figure 3.20):

1. Localize comparisons with a neighborhood graph.
   Ex: minimal spanning tree

2. Eliminate inconsistent adjacencies.
   Ex: neighbors too far away

3. Find connected components in resulting graph.
   Ex: two separate curves

By first localizing computations based on the geometry of the domain space, this process avoids comparing distant nodes, utilizing the physical knowledge of locality and continuity evidenced in distributed physical data. The computational cost is proportional to the size of the underlying neighborhood graph, so this algorithm can be quite efficient.
Figure 3.20: Efficient classification process. (a) Localize comparisons with a minimal spanning tree. (b) Eliminate adjacencies for neighbors that are too far away. (c) Identify connected components as equivalence classes.

Other classifiers can be layered over this basic classification mechanism. For example, classifiers corresponding to standard merging and splitting algorithms in computer vision could repartition the results of other classifiers. A merging classifier would merge adjacent, similar-enough classes, while a splitting classifier would reclassify internally inconsistent classes. Another powerful classifier could test a parameterized equivalence predicate over a range of parameters and uncover the persistent equivalence classes.

3.2.3 Means of Abstraction

SAL provides the abstractor mechanism to connect groups of objects at a lower level of abstraction with single objects at a higher level of abstraction. For example, an equivalence class of points can be abstracted as a region bounded by the convex hull of the points (Figure 3.21(a)), and a set of connected points can be abstracted as a curve (Figure 3.21(b)); different implementations support such processes.

Definition 10 (Abstractor) An abstractor $A = (S, L, H, a)$ where

- $S$ is a space.
Figure 3.21: Examples of abstraction. (a) A set of points is abstracted as a polygon bounded by the set’s convex hull. (b) A set of points is abstracted as a curve based on adjacencies in the neighborhood graph.

- $L \in 2^S$ is a partitioning of $S$.

- $H$ is a space of high-level objects.

- $u : L \rightarrow H$ maps a space of low-level objects to a single high-level object.

An abstractor constructs and maintains the bi-directional mapping, so that it can answer queries about how a group of lower-level objects has been abstracted, or from what group of lower-level objects a higher-level objects was abstracted (Table 3.11).

The higher-level objects produced by abstraction form a more compact description of the data; for example, a spline requires only a few parameters, compared with the original set of points. They also support additional reasoning; for example, a region has area and a curve has curvature, while sets of points do not. Finally, properties that are global with respect to lower-level objects become local with respect to the higher-level objects. For example, in Figure 3.22, the global property of whether a point is inside or outside a collection of curves is hard to calculate, while the local property of whether a point is inside or outside a more abstract region is much easier.
- **high_level_objects** : (Abstractor a) → (Space s)
  - Returns the results of a's abstraction.
  - \( s = H[a] \).

- **localize** : (Abstractor a, Spatial_Object o) → (Space s)
  - Returns the space that o abstracts.
  - \( s \in L[a] \) such that \( u[a](s) = o \).

- **low_level_classes** : (Abstractor a) → (Spaces l)
  - Returns the classes abstracted by a.
  - \( l = L[a] \).

- **redescribe** : (Abstractor a, Space s) → (Spatial_Object o)
  - Returns the abstraction of the space.
  - \( o = u[a](s) \) for \( s \in L[a] \).

Table 3.11: Major abstractor operations.

Figure 3.22: Global/local properties illustrated with the inside-outside problem.
• Primitive: discrete *Spatial Object*.
  - *Cell Complex*: topological structure.
  - *Geometric Object*: properties with respect to *Metric Space*.

• Compound: structured collections of primitive objects.
  - *Space*: collection of objects; *Metric Space* encodes distance measure.
  - *Field*: mapping from domain to feature; encodes continuity.
  - *Ngraph*: mapping from object to neighbors; encodes locality.
  - *Classifier*: partition of objects; predicate encodes similarity.

• *Abstractor*: compound to/from higher-level primitive.

Table 3.12: Summary of SAL programming elements.

The construction of cell complexes is a form of abstraction. For example, the cell complex for a triangle can be built by providing a sequence of three connected edges and redescribing them as a triangle complex.

In abstracting objects with associated features, multiple abstraction processes can be invoked and the resulting higher-level objects paired up. For example, sets of points in a temperature field can be abstracted as regions, the associated temperatures can be abstracted by an average, and the higher-level regions and average temperatures can be associated.

### 3.2.4 SAL Programming Elements Summary

Spatial Aggregation Language programming elements support distributed data analysis, modeling, and control tasks with a variety of powerful, generic data types and operators that encode task-specific knowledge. Table 3.12 summarizes these elements and the knowledge they encode.
3.3 Spatial Aggregation Language Implementation

The Spatial Aggregation Language implementation comprises a C++ library and an interpreted, interactive environment layered over the library. The library supports construction of efficient C++ programs with access to a large set of data type implementations and operations supporting a SAL programming style. The interpreter supports rapid prototyping of application programs by providing a convenient, high-level interface to some of the main data type implementations and operators of the SAL library. Programmers can conveniently explore trade-offs in the specification of domain knowledge such as neighborhood relations and equivalence predicates, interactively examining and modifying the results without having to recompile a program. Graphical inspection tools support manipulation and exploration of the structures in physical data.

3.3.1 SAL C++ Library

The library specifies implementation-dependent functionality for the SAL data types (field, ngraph, etc.) in interface classes, and provides a number of concrete implementation classes (Delaunay triangulation, minimal spanning tree, etc.) meeting that functionality in different ways. It also provides additional operators layered over the interface operators, supporting implementation-independent functionality.

Table 3.13 summarizes some of the data type implementations currently provided by the library. Appendix A provides more detailed descriptions of the library components, and Appendix B discusses implementation details.
• Spatial Object
  - Cell complex: 0-d point, 1-d segment, 1-d line, 2-d polygon, 2-d triangulation, 3-d polyhedron.
  - Geometric properties: corresponding coordinates, length, area, centroid, volume.

• Space
  - Collection: set, sequence, vector.
  - Euclidean coordinate space, indexed in array or k-d tree.
  - Metric space for complexes based on centroid distances.

• Field: scalar field, vector field.

• Ngraph: triangulation, minimal spanning tree, relative neighborhood graph, regular grid, k-nearest neighbors.

• Classifier
  - Clustering mechanism: best match, merging, transitive closure.
  - Equivalence predicate: feature bin, feature distance.

• Abstractor: bounding box, convex hull, curve from connected points in ngraph.

Table 3.13: Summary of SAL C++ library.
3.3.2 SAL Interpreter

The SAL interpreter provides a high-level, function-based interface to the SAL C++ library. The interpreter indexes the names and type signatures of a number of the operations in the library, and can easily be extended to support additional library operations. It allows a programmer to invoke these operations on data and save the results in variables. It performs type checking to ensure that library calls are safe. Finally, it supports passing user-defined, interpreted functions to library operations.

The syntax of the interpreter is straightforward; for the most part, a program simply consists of calls to library functions and assignment of the results to variables. Table 3.14 gives a few simple examples and Table 3.15 details the core of the syntax. Multiple commands on a single line are separated by semicolons. The interpreter includes standard arithmetic and boolean expressions in a C-like syntax. The conditional expression is similar to C's but requires braces; it can be used as part of an expression. Function calls can pass arguments by position or by name; this is especially useful for optional arguments. A user-defined function is a special type of expression introduced by the keyword "function," followed by a parenthesis-enclosed argument list (possibly empty) and a brace-enclosed body. Arguments are lexically scoped. Functions can be used anonymously. A special piece of syntactic sugar wraps up an anonymous function for use in a set-notation style call by specifying arguments as members "in" some collection.

In addition to the command-line interpreter, the SAL environment provides graphical inspection tools for exploring spatially distributed physical data and the extracted structures. The inspectors plot geometric objects, fields, neighborhood graphs, and equivalence classes. They allow the user to select objects, edit fields, and highlight
// Function calls and assignment to variables
a = foo()
bar(a)

// Multiple calls on a line separated by semicolons
a = foo(); bar(a)

// Arithmetic and boolean expressions
a = 3+4 * 5; b = (5 < 6) && false

// Conditional results
c = (if (a < b) { a } else { b })

// Function calls with positional or named arguments
aggregate(points, make_mesh_delaunay())
aggregate(g=make_mesh_delaunay(), s=points)

// User-defined function
max = function(x,y) {
  if (x < y) { y } else { x }
}

// Anonymous function
inc_numbers = map(numbers, function(n) { n+1 })

// Set-notation anonymous function
inc_numbers = map(n in numbers, { n+1 })

Table 3.14: Examples of SAL interpreter use.
Table 3.15: Core SAL interpreter syntax. With the exception of operators (=, +, etc.), language tokens are underlined. Lexical constants are in italics.
the neighbors of an object. The tools are integrated with the environment, so that a selected object can be manipulated by code, and code can modify the current inspection.

Figure 3.23 shows a screen dump of an interaction with the interpreter. The main window has been used to evaluate SAL application code, and has generated a number of graphical inspection windows showing various views of the data and its structures.

This interactive environment supports rapid prototyping of SAL applications by allowing the user to quickly and easily check the results of different instantiations of domain knowledge. Appendix B discusses additional implementation details.

3.4 Example Programs

This section demonstrates how programs can be written using the SAL components. It presents three sample SAL programs: dynamical system analysis in the spirit of KAM [83], vector field analysis in the spirit of phase space control [90], and a boundary tracing operation as in Ullman's visual routines [76]. These examples have been programmed with the SAL interpreter discussed in the previous section.

3.4.1 Dynamical System Analysis

The trajectory bundling application, introduced in Section 3.1, performs some of the work that KAM does in interpreting dynamical systems. It takes as input a set of sampled points in a phase space, groups them into trajectories, and then groups the trajectories into bundles with the same limit behavior.

The first level (Figure 3.24 and Table 3.16) starts by reading the points space of sample points.
Figure 3.23: The SAL interpreter in action: interactive code evaluation and graphical inspection of results.
One of the insights exploited by KAM is that a minimal spanning tree (MST) groups points into a structure similar to curves, but with a few "too-long" edges crossing the curves. To program this in SAL, the points are aggregated into the `point_trig` Delaunay triangulation and `point.mst` derives its minimal spanning tree.

Now `point_classifier` builds equivalence classes by transitively following an equivalence predicate through the `point.mst`. The `good_adj` equivalence predicate tests if two points are close enough compared to other point-point distances nearby in the neighborhood graph. For a given pair of adjacent points, the function first derives the `nearby_ngraph` subgraph with points reachable from the given points in a specified number of steps. It then finds the average and standard deviation of node-neighbor separation among the adjacencies in this graph and checks whether or not a given adjacency is short enough compared to that.

The abstraction jump to the second level (Figure 3.25 and Table 3.17) is made by redescribing the equivalence classes of points as trajectory objects, where a trajectory is a single geometric object with structure determined from a path in the ngraph. The redescription function returned by library function `make_path_to_curve_redescriber` performs such a redescription for a set of points relative to a specified neighborhood graph (here, `point.mst`). The `points_to_traj` abstractor maintains the mapping from classes of points in `point.classifier` to the higher-level `trajs`.

A similar process of aggregating and classifying trajectories can be performed at the higher level. First the trajectories are aggregated into `traj_ngraph` with a substructure aggregator: a trajectory's neighbors are those trajectories with points that were neighbors of the trajectory's points in an underlying ngraph (here, `point.mst`).
Figure 3.24: Example steps in first level of SAL implementation of trajectory bundling application. (a) Input points. (b) Points aggregated into a Delaunay triangulation. (c) The minimal spanning tree of the triangulation. (d) Points joined by short-enough edges.
// (a) Read input points.
points = read_points(infile)

// (b) Aggregate into a Delaunay triangulation.
point_trig = aggregate(points, make_mesh_delaunay())

// (c) Find the minimal spanning tree.
point_mst = mst(point_trig)

// (d) Classify, keeping close-enough neighbors.
good_adj = function(adj) {
  nearby_ngraph = reachable_ngraph(point_mst, adj, depth)
  nearby_edges = faces(make_ngraph_geom(nearby_ngraph), 1)
  edge_lengths = map(nearby_edges, volume)
  m = average(edge_lengths)
  sigma = stddev(edge_lengths, m)
  distance(from(adj), to(adj)) < m + num_hexes*sigma
}

point_classifier =
classify(points,
    make_classifier_transitive(point_mst, good_adj))

Table 3.16: SAL code for first level of trajectory bundling application.
This substructure-based neighborhood graph, inspired by the strong/weak adjacency mechanism of Huang [38], leverages lower-level locality to define higher-level locality.

The same_limit equivalence predicate defines equivalence of trajectories by comparing tangent vectors at corresponding endpoints. Correspondences are defined here by the structure of the point triangulation. For example, in Figure 3.26, point A in one trajectory is connected to points B and C in the neighboring trajectory. The tangent of the first trajectory at point A is compared with the average of the tangents of the second trajectory at points B and C. Traj_classifier transitive follows this equivalence predicate through traj_ngraph.

3.4.2 Dynamical System Analysis Exploration

This section revisits the trajectory bundling example of the previous section, assuming that the user knows the general framework for solving the problem — group points into trajectories into bundles — but doesn’t know how to code it in SAL. The exploration uses a specific input (Figure 3.24(a)) to derive code that not only achieves the desired output (Figure 3.25(c)) for the concrete example but also encapsulates a more general mechanism appropriate for other inputs. The example code steps demonstrate the use of SAL in exploring the implementation space for a problem.

The first step is to group points into trajectories. In the SAL framework, grouping is accomplished by local comparisons between adjacent objects in a neighborhood graph. But what neighborhood graph is appropriate (refer to Figure 3.27 and Table 3.18)?
Figure 3.25: Example steps in second level of SAL implementation of trajectory bundling application. (a) Equivalence classes redescribed as trajectories. (b) Trajectories aggregated based on adjacencies of constituent points. (c) Equivalence classes of trajectories with similar-enough limit behavior.
// (a) Redescribe point classes as trajectories.
points_to_traj =
    redefine(classes(point_classifier),
        make_redefine_op_path_nline(point_mst))
trajs = high_level_objects(points_to_traj)

// (b) Aggregate trajectories based on point ngraph.
traj_ngraph =
    aggregate(trajs,
        make_ngraph_connected_substructure(point_mst,
            points_classifier, 
            points_to_traj))

// (c) Classify based on similarity of tangent vectors.
same_limit = function(adj) {
    t1 = from(adj); t2 = to(adj)
    p1a = end1(t1); p1b = end2(t1)
    t2_points = localize(points_to_traj, t2)
    t2_corresponding_p1a =
        intersection(t2_points, neighbors(point_trig, p1a))
    t2_corresponding_p1b =
        intersection(t2_points, neighbors(point_trig, p1b))
    if (size(t2_corresponding_p1a) == 0 ||
        size(t2_corresponding_p1b) == 0) {
        false
    } else {
        tan_1a = tangent(t1, p1a); tan_1b = tangent(t1, p1b)
        tan_2a = space_centroid(map(p in t2_corresponding_p1a,
            { tangent(t2, p) }))
        tan_2b = space_centroid(map(p in t2_corresponding_p1b,
            { tangent(t2, p) }))
        (dot(tan_1a, tan_2a) >= angle_thresh &&
        dot(tan_1b, tan_2b) >= angle_thresh)
    }
}
traj_classifier =
    classify(trajs,
        make_classifier_transitive(traj_ngraph, same_limit))

Table 3.17: SAL code for second level of trajectory bundling application.
Figure 3.26: Corresponding points of trajectories based on triangulation.

```
// (a) Neighbors within radius of 5
point_nearness_ngraph_5 = aggregate(points, make_ngraph_near(5))

// (b) Neighbors within radius of 2
point_nearness_ngraph_2 = aggregate(points, make_ngraph_near(2))

// (c) 2-nearest neighbors
point_2_nearest_ngraph =
    sub_k_nearest_ngraph(point_nearest_ngraph_5, 2)

// (d) MST
point_mst = mst(aggregate(points, make_mesh_delaunay()))
```

Table 3.18: SAL code for computing point neighborhood graphs during exploration of trajectory bundling application.
Figure 3.27: Point neighborhood graphs during exploration of trajectory bundling application. (a) Radius of 5. (b) Radius of 2. (c) 2-nearest neighbors. (d) Minimal spanning tree.
A simple answer is to try a nearness neighborhood graph, `point_nearness_graph.5`, with a fixed neighborhood radius of 5. This neighborhood graph has too many adjacencies near the center of the space. On the other hand, reducing the radius to 2 for `point_nearness_graph.2` yields too few adjacencies near the edges of the space. Another possibility is to use a variable radius neighborhood graph, where the radius depends on the density of points. However, this would require a mechanism for computing density and appropriately setting the number of neighbors.

Examination of the structure of these two neighborhood graphs reveals a few places where points are connected linearly, as desired for trajectory curves. The key insight highlighted by such linear chains of points is that points tend to be closer to other points in the same curve than to points in other curves. Furthermore, a linear chain connects each point with one or two adjacent points. This insight drives another attempt at choosing a good neighborhood graph: aggregate the points with a \( k \)-nearest neighbors relation, where \( k = 2 \). `Point_2_nearest_graph` keeps each point's two nearest neighbors from the base `point_nearness_graph.5` neighborhood graph.

Now points are connected in a structure very similar to trajectory curves, except for a few possible trouble spots, illustrated in more detail in Figure 3.28:

1. Point \( A \) only has one natural nearest neighbor. Forcing it to have two neighbors creates a "forward adjacency" from it to \( C \), a neighbor's neighbor.

2. Points \( G \) and \( I \) have "cross adjacencies" connecting them to points (\( L \) and \( J \), respectively) that should be in another trajectory curve.

3. Due to varying point density, points \( E \) and \( F \) do not have a symmetric adjacency. In an even more extreme case, they could be disconnected.
Figure 3.28: Potential problems in using k-nearest-neighbors ngraph for trajectory bundling.

Instead of addressing each of these problems individually in an ad hoc manner, the insight utilized by the original program and apparent in this example is that a tree structure is even closer to the desired trajectory curves — it has no forward adjacencies and is connected. `Point_mst` derives such a tree structure as the minimal spanning tree of a Delaunay triangulation (the tree could be found in a fully-connected graph, but the triangulation is more efficient.)

The minimal spanning tree still leaves unsolved the problem of cross adjacencies connecting points that should be in separate trajectory curves. Figure 3.29 and Table 3.19 explore possible methods to eliminate the cross adjacencies. Recall the insight driving the choice of the 2-nearest neighbor graph: points in a curve tend to be closer than points in separate curves. Thus the approach taken here is to eliminate cross adjacencies based on length.

A first attempt at such a filter is to fix a threshold and keep only pairs of points separated by no more than that distance. `Filter_length_3` tries a threshold of 3 units.
Figure 3.29: Filtered point neighborhood graphs during exploration of trajectory bundling application. (a) Points no further away than 3 units. (b) Points no further away than the average distance to their neighbors.
// (a) Eliminate adjacencies between points further away
// than 3 units.
filter_length_3 = filter_ngraph(adj in point_mst, {
  distance(from(adj), to(adj)) <= 3
})

// (b) Eliminate adjacencies between points further away
// than the average distance to their neighbors.
below_average = function(adj) {
  f = from(adj); t = to(adj)
  df = average(map(n in neighbors(point_mst, f),
    { distance(n, f) }))
  dt = average(map(n in neighbors(point_mst, t),
    { distance(n, t) }))
  distance(f, t) < 0.5*(df+dt)
}

filter_average = filter_ngraph(point_mst, below_average)

Table 3.19: SAL code for filtering point neighborhood graph during exploration of
trajectory bundling application.
It successfully eliminates some of the too-long adjacencies, but not all of them; tightening the threshold would eliminate more, but might also eliminate some of the good adjacencies. As with nearness neighborhood graphs, a fixed global threshold is unsuitable here since point density varies.

A variable threshold for point separation can be established by comparing the distance between two points to the average distance between those points and any of their neighbors. If the distance is less than the average, then the adjacency is considered to connect points within a single trajectory; if it is longer than the average, then the adjacency is considered to connect points that should be in separate trajectories. Filter.average uses this heuristic to filter point.mst. While the heuristic almost works, it runs into problems on some adjacencies at the leaves of the tree — if only two adjacency lengths are compared, one must be above average. The heuristic is too local, and the code presented in the last section (Table 3.16(d)) generalizes it by comparing adjacency lengths within some number of steps in the neighborhood graph.

After redescribing connected points as trajectories as in the original code (Table 3.17(a)), the next task is to group trajectories with similar limit behavior (Figure 3.30 and Table 3.20). First, computation must be localized by aggregating trajectories with some neighborhood relation. While the fixed radius neighborhood graph traj.near.5 is easy, it doesn't connect all pairs of trajectories whose limit behavior should be compared. On the other hand, a larger radius might connect pairs of trajectories that should not be compared, since other trajectories lie between them. A better approach, identified by Huang [38], is to leverage the structure of the underlying point.mst: if two points are neighbors in the minimal spanning tree, then consider
the two trajectories to which they belong to be neighbors. Table 3.17(b) uses this approach.

Now the limit behavior of adjacent trajectories can be compared. The basic idea is to compare tangent vectors at corresponding endpoints, but a key implementation detail is to define a method for finding corresponding endpoints. The simplest approach, as in same_end_dirs, is to rely on a corresponding ordering of vertices in the trajectory curves, such that the first end of one is closest to the first end of the other. While this approach happens to work for this problem, it clearly is not general. The code of Table 3.17(c), illustrated in Figure 3.26, uses a more sophisticated method.
// (a) Aggregate trajectories with a fixed radius between centroids.
traj_nearest_nbrgraph_5 = aggregate(trajs, make_nbrgraph_near(5))

// (b) Compare end1 of the one trajectory with end1 of the other,
// and end2 of the one with end2 of the other.
same_end_dirs = function(adj) {
  t1 = from(adj); t2 = to(adj)
  p1a = end1(t1); p1b = end2(t1)
  p2a = end1(t2); p2b = end2(t2)
  tan_1a = tangent(t1, p1a); tan_1b = tangent(t1, p1b)
  tan_2a = tangent(t2, p2a); tan_2b = tangent(t2, p2b)
  (dot(tan_1a, tan_2a) >= angle_similarity &&
   dot(tan_1b, tan_2b) >= angle_similarity)
}
traj_classifier =
  classify(trajs,
    make_classifier_transitive(traj_nbrgraph, same_end_dirs))

Table 3.20: SAL code for classifying trajectories during exploration of trajectory bundling application.

due to Huang [38]. The insight here is that a Delaunay triangulation on the constituent points links vertices of trajectory curves. This supports robust computation of corresponding endpoints for comparison of limit behavior.

3.4.3 Vector Field Analysis

The vector field analysis application interprets a vector field (e.g. wind velocity vectors or temperature gradient vectors) as a set of streamlines (paths through the field following the vector directions). It takes as input a discretized 2-D to 2-D vector field and groups neighboring almost-collinear vectors into streamlines. These equivalent vectors are uncovered by filtering a base ngraph, keeping edges that have similar directions to the vectors, and finding best matches forward and backward along
the vector directions. The resulting streamlines could then be analyzed following the approach for trajectory curves in the preceding example.

The first part of the analysis is shown in Figure 3.31 and Table 3.21). The first step is to read the *vect_field* input field, which maps points (locations) to points (vector directions). Assume that the input field has normalized vector directions (i.e. unit length), so that magnitude is not a factor.

Computation is localized with the *point_ngraph* neighborhood graph. For this example, an 8-adjacency ngraph (all points within a radius of 1.5) suffices; a neighborhood graph with more neighbors would yield potentially smoother curves, since a vector has more potential targets, but it would require more computation.

Now the forward-direction comparison begins by filtering the base *point_ngraph* into the *forward_ngraph* that keeps only adjacencies with directions similar enough to the average vector directions connected by the adjacencies. The function *angle* measures the angle between the vectors and the edge by taking the dot product of the normalized sum of the vectors and the normalized difference between the two point locations.

Among a point's adjacencies with directions similar enough to its vector direction, the neighbor with the most similar vector direction is chosen for the *best_forward_ngraph*. The *forward_metric* function measures similarity by maximizing the angle similarity, penalizing for distance.

The *best_forward_ngraph* links each vector to the neighbor to which it most wants to flow. This creates some "junctions" into which multiple vectors flow. To form single curves, a choice of the best "from neighbor" must be made at each junction.
Figure 3.31: Example steps in first part of SAL implementation of vector field analysis application. (a) Input vector field. (b) 8-adjacency neighborhood graph. (c) Forward neighbors. (d) Best forward neighbors.
// (a) Read vector field.
vector_field = read_point_point_field(infile)
points = domain_space(vector_field)

// (b) Aggregate with 8-adjacency.
point_ngraph = aggregate(points, make_ngraph_near(1.5))

// (c) Compare vector directions with node-neighbor direction.
angle = function (p1, p2) {
    dot(normalize(add(feature(vector_field, p1), feature(vector_field, p2))),
         normalize(subtract(p2, p1)))
}

forward_ngraph = filter_ngraph(adj in point_ngraph, {
    angle(from(adj), to(adj)) > angle_similarity
})

// (d) Find best forward neighbor, comparing vector direction
// with ngraph edge direction and penalizing for distance.
forward_metric = function (adj) {
    (angle(from(adj), to(adj)) -
     distance_penalty * distance(from(adj), to(adj)))
}

best_forward_ngraph =
    best_neighbors_ngraph(forward_ngraph, forward_metric)

Table 3.21: SAL code for first part of vector field analysis application.
// (a) Find backward neighbors by inverting best forward neighbors.
backward_ngraph = invert_ngraph(best_forward_ngraph)

// (b) At junctions, keep best backward neighbor using metric
// similar to that for best forward neighbors.
backward_metric = function (adj) {
    (angle(to(adj), from(adj)) - 0.01*distance(from(adj), to(adj)))
}

best_backward_ngraph =
    best_neighbors_ngraph(backward_ngraph, backward_metric)

// (c) Jump an abstraction level by turning remaining groups
// of adjacencies into curves.
final_ngraph = symmetric_ngraph(best_backward_ngraph,
    extend=true)

point_classes =
    classify(points, make_classifier_transitive(final_ngraph))

points_to_curves =
    redescribe(classes(point_classes),
        make_redescribe_op_path_nline(final_ngraph))

trajs = high_level_objects(points_to_curves)

Table 3.22: SAL code for second part of vector field analysis application.

Figure 3.32 and Table 3.22 illustrate the second part of the vector field application. This process proceeds the same as in the forward direction, but with the neighborhood graph inverted so that flows are in the opposite direction. Backward_ngraph forms the inverse ngraph of best_forward_ngraph, and best_backward_ngraph finds the best match in a manner similar to the formation of best_forward_ngraph.

Now the remaining adjacencies can be turned into curves by making them symmetric (final_ngraph) and classifying and redescribing as in the trajectory bundling example.
Figure 3.32: Example steps in second part of SAL implementation of vector field analysis application. (a) Ngraph inverted from best forward neighbors. (b) Best backward neighbors. (c) Resulting adjacencies redescribed as curves.
3.4.4 Boundary Tracing

Boundary tracing, a basic operation in image analysis, identifies and groups boundary segments from the same object [76]. For example, consider a bitmap of overlapping 2D objects (see Figure 3.33). A boundary tracing operation follows along the curves of the objects, using collinearity to maintain the correct boundary at the "junctions" where boundaries meet. A Spatial Aggregation specification for this example was presented in [86]; this section presents an implementation in the Spatial Aggregation Language.

The algorithm first separates pixels by value and by the number of neighbors of the same value. In particular, it only considers foreground pixels (in this case, those with value 1), and it distinguishes junction pixels that have more than two neighbors of the same value. Equivalence classes of pixels are redescribed as curves, and collinear curves are grouped at the next aggregate level.

The first level (Figure 3.34 and Table 3.23) starts by reading the bitmap input field mapping points to binary values. The foreground points (those with value 1 in the bitmap field) are filtered into \( fg \), and then indexed in the \( points \) derived space.

The points are aggregated into \( point_{ngraph} \), which establishes a 4-adjacency relation by considering each point to be adjacent to other points within a radius of 1 (the spatial index makes this computation efficient). The notion of 4-adjacency, like that of 8-adjacency, is common in image processing and is useful when data points are on a regular grid. In this case, an efficient comparison mechanism need only compare a pixel value with other pixel values touching it. Note that since the background points have been eliminated, some of the 4-adjacent neighbors might not exist.
Figure 3.33: A bitmap containing two overlapping rectangular boundary objects. A boundary tracing operation can separate the two objects.

*Junctionless\_ngraph* is formed as a subgraph of *point\_ngraph* eliminating junctions (nodes with more than 2 neighbors). The connected components in the remaining ngraph are grouped into equivalence classes by *point\_classifier*; since the classes are based solely on connectivity, no additional equivalence predicate is required.

To jump to the second level (Figure 3.35 and Table 3.24), the same mechanism as in the trajectory bundling example abstracts equivalence classes of points as curves in the *points\_to\_curve* abstractor.

*Curve\_ngraph* aggregates curves whose endpoints are connected to the same junction. This is programmed by extending the point ngraph one step, so that each point's neighbors also include its neighbors' neighbors. This effectively skips over the junctions, directly connecting a point on one side of a junction to a point on the
Figure 3.34: Example steps in first level of SAL implementation of boundary tracing application. (a) Points on boundaries of two overlapping objects. (b) Points are aggregated with the 4-adjacency relation. (c) Junctions are removed from the ngraph. The remaining adjacencies form equivalence classes.
// (a) Read bitmap.
bitmap = read_point_point_field(infile)

// Filter and index foreground points.
fg = filter(p in domain_space(bitmap),
   { feature(bitmap, p) == 1 })
points = make_coord_space_2d_buckets(fg)

// (b) Aggregate with 4-adjacency.
point_ngraph = aggregate(point_space, make_ngraph_near(1))

// (c) Remove junctions; the remaining connected components
// are the equivalence classes.
not_junction = function (p) {
   size(neighbors(point_ngraph, p)) <= 2
}
junctionless_ngraph = subgraph(point_ngraph, not_junction)

point_classifier =
   classify(fg, make_classifier_transitive(junctionless_ngraph))

Table 3.23: SAL code for first level of boundary tracing application.
other side. Then the substructure neighborhood graph aggregates curves based on connections of their endpoints in this extended neighborhood graph.

Finally, \textit{curve\_classifier} groups adjacent collinear curves into coherent boundaries. The \textit{collinear} equivalence predicate examines the structure of a pair of curves, comparing the points and the corresponding tangent vectors at their ends. Curves are considered equivalent if the tangents at neighboring endpoints are similar enough in angle.

3.5 Programming Style

The Spatial Aggregation Language supports a particular style of programming for data interpretation tasks. This style can be summarized as follows:

\textit{Use knowledge of locality, continuity, and spatio-temporal scales to extract and exploit multi-layer structures in spatially distributed physical data.}

This section discusses in more detail what that means and how to approach a task in this style. In particular, it examines the types of applications for which SAL is appropriate and how to design and implement SAL programs. The example programs in the last section exemplify the SAL style, and the trajectory bundling and boundary tracing applications will be used as running examples.

When is SAL Appropriate?

Recall that Spatial Aggregation is an instance of \textit{imagistic reasoning}, where computations are structured around perception-like operations on image-like representations of data. A key part in programming a SAL application is to adopt the "imagistic stance" by encoding a task in terms of geometric and topological structures in image-like representations. For example, configuration space-based mechanism...
Figure 3.35: Example steps in second level SAL implementation of boundary tracing application. (a) Point classes are redescribed into curves. (b) Curves are aggregated by nearness. (c) Curves are grouped into boundaries by collinearity.
// (a) Redescribe point classes as curves.
points_to_curve =
    redescribe(classes(point_classifier),
        make_recescribe_op_path_nline(junctionless_ngraph))
curves = high_level_objects(points_to_curve)

// (b) Aggregate curves based on connection to same junction.
point_closure = closure_ngraph(point_ngraph, 2)
curve_ngraph =
    aggregate(curves,
        make_ngraph_connected_substructure(point_closure,
            point_classifier,
            points_to_curve))

// (c) Classify curves based on collinearity.
collinear = function(adj) {
    c1 = from(adj); c2 = to(adj)
    p1a = end1(c1); p1b = end2(c1)
    p2a = end1(c2); p2b = end2(c2)
    tan_1a = tangent(c1, p1a); tan_1b = tangent(c1, p1b)
    tan_2a = tangent(c2, p2a); tan_2b = tangent(c2, p2b)
    ((is_member(neighbors(point_closure, p1a), p2a) &&
        abs(dot(tan_1a, tan_2a)) >= angle_thresh)) ||
        similarly for other pairs
}
curve_classifier =
    classify(curves,
        make_classifier_transitive(curve_ngraph, collinear))

Table 3.24: SAL code for second level of boundary tracing application.
analysis views the configuration of a mechanism as a point with coordinates for the mechanism's degrees of freedom. The analysis then uncovers regions of that space corresponding to feasible configurations of the mechanism.

Adopting this stance yields the following input/output characteristics for SAL-based applications:

- **Input**: spatially-distributed data described as a field.
- **Output**: abstract geometric/topological structures uncovered in the data.

The abstract output then serves as the basis for inferring behaviors, designing controls, and so forth. These task requirements serve to constrain the structure of the desired output, and perhaps even to drive the processing from input to output.

In the trajectory bundling example, the task is to determine qualitatively-similar states. This task is described imagistically by representing states as points in a phase space and behaviors as trajectories and bundles. Thus the input is a set of sample points and the output is a set of curve bundles.

In the curve tracing example, the task is to distinguish boundaries of separate objects. This is already an imagistic problem. The input is a bitmap, and the output is a set of connected boundary curves.

SAL represents data in terms of fields and geometric objects, explicitly utilizing metric properties of the input data. This might not be appropriate for some applications. For example, many qualitative spatial reasoning systems (e.g. [16, 63]) use logic-based representations for applications where only topological properties of the input data are known (e.g. object A is to the right of object B). In contrast,
analogic systems (including SAL) implicitly represent these properties for manipulation on demand [14]. By basing computation on high-level structures grounded in analogic data, SAL can avoid the expensive theorem-proving approaches required by logic-based representations.

In order to bridge the gap between input field and abstract structural description, SAL relies on physical knowledge such as continuity and locality. The input data and underlying physical process must exhibit these properties in order for SAL to be applicable. For example, it is hard to extract structures in a very smooth image, or in a white-noise image.

In summary, SAL is suitable for a wide range of tasks expressible in terms of the discovery of geometric and topological structures in spatially distributed data by exploiting continuity and locality at multiple spatio-temporal scales.

How are Applications Designed?

The goal of SAL application design is to identify the abstraction hierarchy linking the input field to the output description, the types of structures that are to be manipulated at various levels in the hierarchy, and the requirements necessary for jumping from one level to the next. Following are the main steps in specifying these details; an actual design process might iterate, alternating between partial solutions to these steps.

1. Continuity and different spatio-temporal scales give rise to regions of uniformity at multiple levels of abstraction. Identify appropriate structural descriptions for such regions.
In the trajectory bundling example, structures at different scales include sample points, trajectories, and bundles of trajectories. In the boundary tracing example, structures include the pixels in the input bitmap, junctions, curve fragments, and boundary curves.

2. Based on the relationships between these structures, choose abstraction levels. For example, a substructure/structure relation can group points into curves into pipes or points into regions into bodies. An adjacency relation can merge triangles into polygons.

In the trajectory bundling example, points are a substructure of trajectories which are a substructure of bundles. In the boundary tracing example, points can be adjacent to junctions, points are a substructure of curves, and adjacent curves comprise boundaries.

Specification of an abstraction hierarchy can proceed both bottom-up and top-down to bridge the gap between type of input and type of desired output. For example, in the trajectory bundling example, an abstraction hierarchy could be specified bottom-up by noticing that points can be grouped into curves which can be grouped into bundles. Alternatively, the hierarchy could be specified top-down by noticing that bundles are comprised of curves which are comprised of points.

3. Identify how groups of objects at one level are to be redescribed as single objects at a higher level. Identify the sources of uniformity, the preconditions necessary for objects to be grouped, the abstraction transformation to be performed on the groups, and the postconditions that will be true for the higher-level objects.
The sources of uniformity in the trajectory bundling example include nearness of points and similarity in curvature of trajectories. For the abstraction of points into trajectories to be applicable, the points must be linearly connected. For the abstraction of trajectories into bundles to be applicable, the trajectories must be adjacent and have similar limit behavior.

Sources of uniformity in the boundary tracing example include equality of bit value, number of neighbors, and collinearity of curve fragments. Foreground points can be abstracted into curve fragments if they are linearly connected and are not junctions. Curve fragments can be abstracted into boundary curves if they are collinear.

What Programming Discipline is Followed?

A SAL program navigates through the designed abstraction hierarchy by manipulating fields, spaces, neighborhood graphs, and equivalence classes in order to connect the output of a lower level of abstraction to the input of the next higher level of abstraction. The use of these data types imposes a particular discipline on the programs (recall the characterization of SAL applications at the beginning of this section):

- **Distributed data:** SAL applications manipulate spatially distributed data, for example from a set of sensors or from a simulation. The *Space* and *Field* data types package up distributed data and features for collection-based processing, allowing element-wise operations to be distributed out to the members of a collection and global properties to be gathered back from the collection. A programmer can form and manipulate these compounds as first-class objects, selecting sub-spaces and sub-fields and finding associated spaces and fields. This
supports reasoning about interactions and evolutions at the group level, rather than at the individual object level.

- **Locality:** SAL applications exploit the fact that physical interactions propagate from local to global, in order to build programs that reason from local to global. In particular, the $N$graph data type explicates an adjacency relation encoding a domain-specific definition of locality. The adjacencies in the graph support distributed, decentralized processing by allowing interactions and comparisons to be invoked only upon local groups of nodes. Ngraphs, like spaces and fields, allow these interactions to be specified for entire groups of objects, with the local interactions and comparisons distributed out among adjacent objects in the neighborhood graph.

- **Continuity:** To uncover structures in spatially distributed data, SAL applications exploit continuity of fields to find regions of uniformity. The Classifier data type uses an application-specific search mechanism and definition of equivalence in order to find such regions. One particularly efficient, distributed classifier mechanism uses local comparisons in a neighborhood graph to build equivalence classes transitively, linking pairs of neighboring objects that satisfy an application-specific equivalence predicate.

- **Multi-layer structures:** SAL applications use Abstractors to group similar-enough objects into single higher-level objects. The SAL Cell Complex data types support this process by representing discrete spatial objects as structured collections of more primitive spatial objects (faces).
How are SAL Component Implementations Chosen and Manipulated?

The final step in SAL application development is to choose implementations of the SAL data types and instantiate them with appropriate domain knowledge. This process is certainly domain-specific, but it is constrained by the preconditions and postconditions for the abstraction transformation. In particular, the code for one layer must build ngraphs and find equivalence classes within them that satisfy the preconditions for abstraction to the next layer.

The choice of neighborhood relation depends on the use being made of the ngraph, but several possibilities exist:

- Based on desired structure.
  For example, a minimal spanning tree is structurally similar to the desired curves of the trajectory bundling application. A Delaunay triangulation partitions a space into planar regions. A grid encapsulates the structure of regularly sampled input points.

- Based on locality/communication requirements.
  For example, to minimize the amount of computation, compare a pixel with only its 8 adjacent neighbors.

- Based on computational requirements.
  For example, a regular grid is required for solving partial differential equations with finite differences.

- Based on computational complexity.
  For example, with a good spatial index, a $k$-nearest neighbors ngraph might be
cheaper to construct than a minimal spanning tree, and might serve as a good enough approximation.

Equivalence predicates also come in some standard varieties:

- **Proximity of objects.**
  
  For example, close-enough points are grouped into trajectories.

- **Similarity of objects.**
  
  For example, trajectories with similar-enough curvatures are grouped into bundles.

- **Proximity of corresponding feature objects.**
  
  For example, pixels with close-enough intensity values are grouped into regions.

- **Similarity of corresponding feature objects.**
  
  For example, sensors with temperatures that fall into the same bin are grouped into isothermal regions.

In the trajectory bundling example, the point-to-trajectory layer must identify linear chains of points. It first builds a minimal spanning tree, which is structurally similar to a linear chain. It then performs local comparisons in the tree, using an object-proximity equivalence predicate comparing edge lengths. The trajectory-to-bundle layer must identify adjacent trajectories with similar limit behavior. It localizes computation by comparing pairs of trajectories whose substructure was connected in the minimal spanning tree. It then applies an object-similarity equivalence predicate comparing curve curvatures.
In the boundary tracing example, the point-to-curve-fragment layer must identify linear chains of non-junction foreground points. It filters out the background and builds a 4-adjacency graph structurally similar to jagged curves. It filters out the junctions by eliminating points with more than two neighbors, leaving remaining connected points as equivalence classes. The curve-fragment-to-boundary-curve layer must identify collinear curve fragments. It localizes computation by considering pairs of curve fragments whose endpoints are close, and then applies an object-similarity equivalence predicate to group collinear fragments based on endpoint tangent vectors.
CHAPTER 4

DECENTRALIZED CONTROL DESIGN

This chapter presents a case study application of the Spatial Aggregation Language to the design of decentralized controls for thermal regulation. Structures uncovered in fields serve as the basis for algorithms that design control placements and control actions. In particular, this chapter introduces the influence graph mechanism (joint work with Feng Zhao [4, 5]) to encode dependencies between control nodes and field nodes. Influence graph-based design mechanisms support explicit trade-offs between factors such as amount of computation, amount of communication, and resulting control quality. Influence graphs allow explanation of and meta-level reasoning about the resulting designs, in terms of the physical knowledge they represent. While the case study is firmly grounded in the thermal regulation domain, the influence graph abstracts a generic set of reasoning mechanisms for control design problems requiring placement and parametric optimization of decentralized controls for distributed physical fields.

The algorithms in this chapter utilize SAL data types and operators to express important computational patterns in decentralized control design. Where appropriate, this chapter extends SAL with new high-level operators supporting these computational patterns. To emphasize the distributed nature of the algorithms, many of them
are presented graphically, depicting data flow among processing elements. More traditional pseudocode also helps illustrate the high-level computational structure of the algorithms.

The chapter proceeds as follows. First, Section 4.1 describes the application domain. Section 4.2 presents SAL-based techniques for modeling heat flow. Section 4.3 discusses the construction of influence graphs for decentralized control problems. Influence graph-based control design algorithms are elaborated in two separate sections: Section 4.4 presents algorithms for determining control placements, while Section 4.5 presents algorithms for determining control actions. Finally, Section 4.6 discusses the broader applicability of the approaches described in this chapter.

4.1 Introduction

4.1.1 Problem Domain

Consider two different thermal regulation systems, represented in Figures 4.1 [21] and 4.2 [43]. Doumanidis developed the system in Figure 4.1 for rapid prototyping in thermal fabrication (i.e. welding). It includes a servodriven X-Y positioning table, upon which the parts to be joined are placed, a plasma-arc heat source, and an infrared camera providing temperature data. Doumanidis applies feedback control to a linearized model of the system whose parameters are estimated at run-time from temperature distributions [21]. Groups at Stanford and Texas Instruments developed the system in Figure 4.2 for rapid thermal processing for semiconductor curing, where a uniform temperature profile must be maintained to avoid defects. The control strategy is somewhat decentralized, providing separate power zones for three separate rings of heat lamps [43].
Figure 4.1: Rapid thermal processing system welds by moving a part on a positioning table based on feedback from an infrared camera.

Figure 4.2: Rapid thermal processing for semiconductor manufacturing maintains a uniform temperature distribution by independent control to separate rings of heat lamps.
Figure 4.3: Industrial heat treatment of a piece of material. The control objective is to achieve a specified temperature profile over the material by applying heat at a small number of locations, shown as dark circles.

As an abstraction of these applications, this chapter adopts as a running example the generic problem of temperature regulation for a piece of material [39], as shown in Figure 4.3. The temperature distribution over the material must be regulated to some desired profile by a set of distributed point heat sources that can be individually controlled. This thesis only considers the control design problem, assuming that enough temperature data is available (e.g. from an infrared camera). The observability problem can be addressed similarly to the controllability problem; see Section 5.2 for further discussion.

This control design problem extends the above applications, using a set of decentralized controls to achieve the desired temperature profile. Many applications require such decentralized control in order to ensure adaptivity, robustness, and scalability. For example “smart” buildings control temperature with networks of sensors and actuators; decentralized control allows the network to overcome failures in individual control elements and to expand without complexity that increases exponentially with the number of controls. Designing a set of decentralized controls for a distributed physical system requires achieving global control objectives through appropriate combinations of local control actions. The Spatial Aggregation Language supports explicit reasoning about the interactions of controls at multiple scales.
The control design depends on the thermal process in the material and constraints on the design itself. A large design space must be searched subject to structural and performance constraints:

- **Structural constraints:** geometry, physical properties, boundary conditions.
- **Performance constraints:** desired profile, optimality conditions on solutions, restrictions on control sources (placement and strength).

In the thermal control problem, the control objective is to establish a particular temperature distribution over the entire field, using a small set of discrete heat sources, subject to constraints on the maximum source output and acceptable temperature fluctuations. There are two important components to the design: structure design (e.g. the number and location of heat sources) and parametric design (e.g. heat source values). Rather than addressing both design components simultaneously, which would yield a very large and complex design space, the approach taken here is to design the structure first and then design the parameters. The structure is designed in a manner that actually aids the parametric design, by placing controls so that they minimally interfere with each other. This approach is particularly appropriate for applications where the structure design is performed once (to place controls), and the parametric design is performed repeatedly (e.g. for various desired profiles).

### 4.1.2 Overview of the Approach

The following sections use and extend SAL to design controls for thermal regulation. The first task is to model the heat flow in a problem domain. Figure 4.4 overviews some major computational patterns in this process. A geometric object
describing a problem domain (e.g. a piece of material for which the temperature is to be regulated) is discretized into a space of spatial object nodes and an associated neighborhood graph. Local interaction rules in the neighborhood graph support the computation of heat flow. This process can occur at multiple levels of abstraction, with approximations at coarse resolutions driving refinements at finer resolutions. It can also occur separately in subregions of the initial space, with results iteratively combined. Section 4.2 presents SAL encodings of these common computational patterns.

The second task is to design both control placements and control parameters to achieve the desired control objective. SAL operators perform control design based on structural representations of heat flow, as shown in Figure 4.5. Sample control probes allow structural representations of heat flow to be extracted. These control probes are classified based on similarities in the response of the field to their actions. In the example, the geometric constraint imposed by the narrow channel in the dumbbell-shaped piece of material results in similar field responses to the two probes in the left half of the dumbbell and similar responses to the two probes in the right half of the dumbbell. Based on the resulting classes, the field is decomposed into regions to be separately controlled. In this case, the left half of the dumbbell is decomposed from the right half. Controls are placed in the regions and optimized by adjusting their outputs in response to their effects on the field.
Figure 4.4: Overview of SAL-based computational patterns in modeling heat flow.
Figure 4.5: Overview of SAL-based decentralized control design for thermal regulation.
4.2 Modeling Heat Flow

The physical process of heat diffusion is modeled by partial differential equations (PDEs). In particular, this case study focuses on steady-state (asymptotic) temperature distributions, modeled by the Laplace equation:

\[ \nabla \cdot \nabla k \phi + \dot{Q} = 0 \]  

(4.1)

\( \nabla \) is the spatial derivative (e.g. \( \frac{\partial}{\partial x} \vec{i} + \frac{\partial}{\partial y} \vec{j} \) for a 2-D Euclidean space), \( \phi \) is the temperature, \( k \) is material conduction coefficient, and \( \dot{Q} \) is the source value representing heat per unit time and volume. Intuitively, this equation models heat diffusion as a smoothing process, reducing sharp spatial variations in temperature.

Given the description of a piece of material (geometry, boundary conditions, and material properties), solving the heat equation yields the resulting temperature profile over the material. However, for non-trivial descriptions, it is often infeasible to find a closed-form solution to the equation. Thus scientists and engineers turn to simulations on discretized representations in order to extract the behavior of a system.

The following sections discuss Spatial Aggregation Language encodings of common partial differential equation-related numerical routines (refer again to Figure 4.4 in Section 4.1.2 for an overview.)

**Domain Discretization**

The first step in computing heat flow is to discretize the problem domain into a set of representative objects for which the temperature will be solved. For a fairly uniform spatial domain, a grid of points (Figure 4.6(a)) often suffices; for a more complex domain (in geometry or material conditions), a mesh of triangular elements (Figure 4.6(b)) offers better approximation at the expense of higher complexity.
Figure 4.6: Domain discretizations. (a) Regular domain discretized as a set of points. (b) Complex domain discretized as a mesh of triangular elements.

| disaggregate : (Geometric_Object o) → (Space s) |
| Discretizes o into a space of objects. |

Table 4.1: SAL disaggregate operator.

The important operation of forming a discretization for a geometric specification is handled by the new SAL *disaggregate* operator (Table 4.1). Implementations include point discretizers and mesh discretizers, as in Figure 4.6. The disaggregation must choose a “good” discretization that takes into account geometric and physical properties of the domain. Mesh generation packages such as Shewchuk’s Triangle package [69] use sophisticated geometric techniques to satisfy such criteria.

Given a discretization, the temperature distribution is to be computed as a scalar field mapping points to temperature values.
Solving the Diffusion Equation

Discretization of a problem domain allows the task of solving an infinite-dimensional partial differential equation to be converted into that of solving a finite set of coupled ordinary differential equations. Two common methods for deriving such a set of equations are the finite difference method and the finite element method. The finite difference method approximates the spatial derivatives in the heat equation by comparing temperatures among points in a grid (Figure 4.6(a)). The finite element method minimizes an error term derived from conservation of energy over the elements in a mesh (Figure 4.6(b)) [91].

Standard engineering techniques represent the finite difference and finite element equations in matrix form and apply linear equation solving routines. Both methods lead to systems of the following form:

\[ A\phi = s \]  

\[ (4.2) \]

\( A \) is the capacitance matrix formed from the discretization. \( \phi \) is the resulting temperature distribution at the nodes in the discretization, and \( s \) is the contribution from heat sources and boundary conditions.

The capacitance matrix is sparse: entry \((i, j)\) in \( A \) is non-zero if and only if \( \phi_i \) and \( \phi_j \) are neighbors in the discretization. This property makes the system amenable to solution by relaxation methods, which start with a guess at the resulting temperature profile and iteratively improve the guess until the process converges. The original system (4.2) is rewritten into a form with \( \phi \) on both sides of the equation:

\[ \phi = B\phi + u \]  

\[ (4.3) \]
For example, one possible rewriting (Jacobi) sets $B = D^{-1}(L + U)$ and $u = D^{-1}s$, where $A$ has been separated into $A = D - L - U$, with $D$ a diagonal matrix, $L$ a lower-triangular matrix, and $U$ and upper-triangular matrix [10].

Then successive approximations to the solution are formed by using the current value of $\phi$ on the right-hand side of (4.3) to compute a new value of $\phi$ on the left-hand side:

$$\phi^{(1)} = B\phi^{(0)} + u$$
$$\phi^{(2)} = B\phi^{(1)} + u$$
$$...$$
$$\phi^{(n)} = B\phi^{(n-1)} + u$$

The convergence rate of this technique depends on the form of the rewriting from (4.2) to (4.3): common approaches include the Jacobi method, which simultaneously updates all members of $\phi$, the Gauss-Seidel method, which progressively updates elements of $\phi$, and successive overrelaxation, which extends this approach to mix in differing proportions of old and new $\phi$ values [39].

The Spatial Aggregation Language supports programming such field solvers through local interactions among the constituent objects of a discretization. This is a different viewpoint from the matrix-solving mechanism, and it corresponds more closely to the physical intuition about how heat and other similar properties propagate among objects in a field. This makes it a convenient mechanism for programming such solvers and understanding the resulting process. Furthermore, by following a uniform local-interaction methodology, SAL solvers are naturally parallelizable.
Figure 4.7: Neighborhood graphs for diffusion simulation. Filled-circle nodes are the neighbors of the large filled-circle node. (a) Finite difference 4-adjacency graph relating points in a regular grid. (b) Finite element mesh relating a point with other vertices of shared triangular elements.

The first step in programming local interactions in SAL is to aggregate objects into a neighborhood graph establishing an appropriate adjacency relation. In order to estimate temperature gradients, the finite difference method relates a point’s temperature to the temperatures at points along axis-parallel directions. One simple estimate compares temperature against the four adjacent points in a grid, as shown in Figure 4.7(a). In order to minimize an error term over mesh elements, the finite element method relates a point’s temperature with the temperatures at other vertices of the elements to which the point belongs. A standard approach is to use a triangulation, as in Figure 4.7(b).

Given a neighborhood graph localizing computation, the heat equation is solved by repeatedly updating point temperatures based on constraints from temperatures at neighboring points, until very little change occurs on an update. While the finite difference and finite element methods use different update formulas, they share a common computational structure, as shown in Table 4.2 and Figure 4.8. The new SAL relax operator (Table 4.3) encapsulates this computational structure. Different
Until little change occurs on all updates
For each node in a space
Update the corresponding field value.

Table 4.2: SAL algorithmic description of relaxation-based heat equation solution.

Figure 4.8: SAL data flow for relaxation-based heat equation solution: update each point's temperature based on constraints from temperatures at neighboring points.

implementations of the operator impose various orderings on the application of the relaxation formula (e.g. update nodes independently for the Jacobi method or sequentially for the Gauss-Seidel method) and mix in varying proportions of the old value and new value (e.g. for successive overrelaxation).

Multi-Level Solving

Multigrid and multi-level methods [10] utilize the insight that, in order to find a solution to the heat equation for a fine discretization, a good starting point is the solution at a coarser discretization. In particular, to solve the heat equation (4.2)
\[relax \in \text{(Field f.} \\ \ \ \ \ \ \ \ \ \ \ \ \ \text{Spatial_Object} \times \text{Number} \rightarrow \text{Number}) \ \ \ \ \ r,\]
\[\text{(Number} \times \text{Number} \rightarrow \text{Boolean}) \ \ \ \ \ c)\]
\[\rightarrow ()\]

Repeatedly updates each object in the field based on the relaxation formula \( r \), until all values have converged according to test \( c \).

Table 4.3: SAL relax operator.

\[A \phi = s, \text{a multigrid method forms a sequence of increasingly coarse systems:}\]

\[
\begin{align*}
A_0 \phi_0 &= s_0 \\
A_1 \phi_1 &= s_1 \\
&\ldots \\
A_n \phi_n &= s_n
\end{align*}
\]

The coarsest system \((\phi_n)\) is solved first, its solution is used to initialize the relaxation process for the next coarsest system, and the process repeats, using the solution \(\phi_{i+1}\) to initialize the relaxation process for the next finer solution \(\phi_i\).

To program this process in SAL requires the introduction of new operators linking the spaces and neighborhood graphs for fine discretizations with those for coarse ones. The \textit{coarsen} operator moves from fine-grained discretizations to coarse-grained discretizations (i.e. build \(A_{i+1}, \phi_{i+1}, \text{and} s_{i+1} \) based on \(A_i, \phi_i, \text{and} s_i\)); the \textit{refine} operator does it in the opposite direction. For example, a grid-based coarsener could simply uniformly sample a finer grid to produce a coarser grid, as shown in Figure 4.9(a); this could be programmed with a standard SAL classification process. A mesh-based
Figure 4.9: Multi-level coarsening/refining. (a) Uniform grid. (b) Unstructured mesh.

Table 4.4: SAL coarsen and refine operators.

- **coarsen**: (Space $s_1$, Ngraph $g_1$) → (Space $s_2$, Ngraph $g_2$)
  Produces a space and neighborhood graph coarsened from the originals.

- **refine**: (Space $s_1$, Ngraph $g_1$) → (Space $s_2$, Ngraph $g_2$)
  Produces a space and neighborhood graph refined from the originals.

coarsen/refiner would require a more sophisticated scheme (e.g. based on a hierarchy maintained during construction of the mesh \[17\]), as shown in Figure 4.9(b). Table 4.4 lists the syntax for the coarsen and refine operators.

Utilizing the new coarsen or refine operator, along with the previously-discussed relax (Table 4.3) and interpolate (Table 3.7) operators, multi-level solvers are naturally expressible in SAL. Table 4.5 provides a high-level algorithmic description of this process, and Figure 4.10 shows the data flow among objects.

Using results at a coarse level to speed up the solution at a fine level is not restricted to the initial phase of solving. Let $\phi^*$ be the unknown correct solution and
To solve at level $i$:
   If coarse enough, relax.
   Else
     Coarsen, producing level $i + 1$.
     Solve at level $i + 1$.
     Interpolate values for level $i$.
     Relax from that state.

Table 4.5: SAL algorithmic description of multi-level solving.

Figure 4.10: SAL data flow for multi-level approximation-based solving of the heat equation: interpolation of coarse-grained solutions drives the solution of finer-grained levels.
consider the \textit{error} vector:

\[ e = \phi^* - \phi \]  \hspace{1cm} (4.4)

Then by the heat equation, \( A\phi^* = s \), so

\[ Ae = s - A\phi \]  \hspace{1cm} (4.5)

Now define the \textit{residual} vector:

\[ r = s - A\phi \]  \hspace{1cm} (4.6)

Then, combining (4.5) and (4.6),

\[ Ae = r \]  \hspace{1cm} (4.7)

Since (4.6) has the same form as the original heat equation, the same technique of using coarse-grain approximations to drive fine-grained solutions can be applied. In this case, this means that coarse-grained solutions to the error are used to correct fine-grain approximations to the solution. In practice, this lets coarse-grained discretizations solve for the smooth parts of the solution that would take longer to find on fine-grained discretizations.

In programming a solver using this residual error correction scheme, the relaxation process is run for some number of steps, values at the coarse level are initialized with values derived from the residual at the fine level, an interpolation of the coarse-level error solution is used to correct the fine-level solution, and the process is repeated. The same framework as in Figure 4.10 applies for finding the correction, but the values being passed between the levels refer to the residual and error rather than the actual solution.
The SAL specification elucidates the computational structure of multi-level solvers in terms of the powerful and generic SAL data types and operators. Specific details of the solving techniques can then be modularly instantiated in this framework.

**Domain Decomposition**

Domain decomposition techniques for solving the heat equation split a field into (possibly overlapping) pieces, add artificial boundary conditions at the edges of the pieces, separately solve the equation for the pieces, combine the solutions where the pieces meet, and repeat until the separate solutions agree [12]. To be more specific, in order to solve the heat equation (4.2) \( A\phi = s \), a domain decomposition technique forms a set of systems:

\[
\begin{align*}
A_0\phi_0 &= s_0 \\
A_1\phi_1 &= s_1 \\
&\vdots \\
A_n\phi_n &= s_n
\end{align*}
\]

The separate systems are defined for different parts of the original domain. Separate relaxation processes compute solutions \( \phi_0^{(0)}, \phi_1^{(0)}, \ldots, \phi_n^{(0)} \) for the subregions; these are combined where regions overlap. The resulting combination serves as the initial guess for a relaxation processes computing \( \phi_0^{(1)}, \phi_1^{(1)}, \ldots, \phi_n^{(1)} \), and the process repeats.

Again, SAL provides a nice high-level framework in which to program this style of computation, classifying a discretization into separate regions and then applying the previously demonstrated techniques to solve the heat equation on the subregions. The field operator *combine* (Table 3.6) is used to combine the subfields, using values
Classify the domain into regions.
Extract the subfield and subgraph for each subregion.

Until solutions for the regions agree where they meet:
   Separately relax through the subgraphs to solve the subfields.
   Combine the sub-solutions.

Table 4.6: SAL algorithmic description of domain decomposition-based solving.

Figure 4.11: SAL data flow for domain decomposition-based solving of the heat equation: repeatedly combine solutions for separate classes of nodes until they agree where they overlap.

from one or the other where the domains are disjoint, and reconciling values (e.g. by averaging) where they overlap or meet. Table 4.6 provides an algorithm description. Figure 4.11 shows the data flow of the algorithm.

A key step in the domain decomposition algorithm is to choose a good decomposition; that is, to choose the classification of the domain into subregions. One standard approach is spectral partitioning [71], which seeks to minimize the number of edges cut in a (neighborhood) graph. It does this by examining the structure of the graph’s Laplacian matrix, which encodes in its non-zero structure the connectivity between
Let \( e \) be the Fiedler vector of the ngraph.

Classify the nodes with equivalence predicate:
- Compare signs of corresponding values in \( e \).

<table>
<thead>
<tr>
<th>Table 4.7: SAL algorithmic description of spectral partitioning.</th>
</tr>
</thead>
</table>

Points. Specifically, entry \((i, j)\) in the Laplacian matrix has value \(-1\) if and only if there is an edge from node \(i\) to node \(j\) in the graph; entry \((i, i)\) has value equal to the total number of edges from node \(i\). It turns out that a good approximation to the optimal partitioning (minimum number of cut edges) can be achieved by separating nodes according to the corresponding values in the eigenvector for the first non-trivial eigenvalue of this matrix (the Fiedler vector). Intuitively, this is similar to partitioning a one-dimensional domain by looking at the sign of a sine wave stretched over the domain [19].

This partitioning process fits naturally into the SAL framework. Using neighborhood graph edges to establish a Laplacian matrix and then classifying nodes based on values in the Fiedler vector. Table 4.7 and Figure 4.12 detail the SAL-based spectral partitioning algorithm.

### 4.3 Influence Graph

In order to design decentralized controls for a physical field, it is necessary to reason about the effects of the controls on the field. The Spatial Aggregation Language provides operators and data types for extracting structural descriptions of a field. However, these operators and data types encode only the structure of the field,
Figure 4.12: SAL data flow for spectral partitioning: form equivalence classes based on values in the first non-trivial eigenvector of the Laplacian matrix for the neighborhood graph.

not the structure of the relationship between the controls and the field. This section introduces the *influence graph* mechanism to represent such dependencies. Influence graphs are computed and manipulated with operators layered over the standard SAL operators introduced in the previous chapter. By explicitly representing the dependencies of field nodes on control nodes, influence graphs support the control design techniques discussed in the rest of this chapter.

### 4.3.1 Thermal Hill

A heat source influences the temperature distribution in a field through heat propagation. Figure 4.13(a) shows that the steady-state influence of a source on a field forms a "thermal hill": the temperature decays away from the source. When multiple sources affect a thermal field, their thermal hills interact, jointly affecting the temperature distribution (Figures 4.13(b) and Figure 4.13(c)). Figure 4.14 shows a hill in a more complex domain.
Figure 4.13: Steady-state thermal hills around sources. The vertical axis represents temperature value. (a) A single source. (b) Two fairly independent sources. (c) Two tightly coupled sources.

Figure 4.14: Thermal hill for a source on a complex domain.
The structure of these thermal hills exposes quite a bit about the influence of a heat source on the temperature field. Temperature decays away from the source at different rates in different directions, due to different constraints from geometry, boundary conditions, and material properties. Similarly, thermal hills from heat sources at different locations have different shapes. Thermal hills expose the locality of the effects of a heat source: a heat source strongly affects nearby field nodes and only weakly affects further away field nodes, depending of course on the conduction properties of the material.

In order to take advantage of these and other properties at a high level of abstraction, influence graphs serve as an abstract, domain-independent representation encoding this knowledge.

4.3.2 Influence Graph Construction

An influence graph records the dependencies between control nodes and spatial objects in a field as edge weights in a graph.

Definition 11 (Influence Graph) An influence graph is a tuple \((F, C, E, w)\) where

- \(F\) is a set of field nodes.
- \(C\) is a set of control nodes.
- \(E = C \times F\) is a set of edges from control nodes to field nodes.
- \(w : E \rightarrow \mathcal{R}\) is an edge weight function with \(w((c, f))\) the field value at \(f\) given a unit control value at \(c\).

Hence, the graph edges record a normalized influence from each control node to each field node. The thermal hills in the last section (e.g. Figure 4.13(a)) are pictorial
representations of the edge weights for an influence graph from one heat source to the nodes of a temperature field.

An influence graph is constructed by placing a control with unit value at each control location of interest, one at a time, and evaluating the field at field node locations of interest. The method of evaluation is problem-specific. For example, it could be found by numerical simulation (e.g. using the relaxation processes of the previous section), experiment, or even explicit inversion of a capacitance matrix. An influence graph, like other SAL data types, then serves as a high level interface caching important information about spatially distributed physical systems; in this case, the information indicates dependencies of field values upon controls.

Influence graph operators manipulate the dependency information; for example, to find the influences from a control node (i.e. the field or hill resulting from that node) or the influences to a field node (i.e. the effects of the different controls). Table 4.8 details these operations. The standard SAL operators are useful in both the control space and the field node space: for example, neighborhood graphs of control nodes can be used to program local interactions among controls, and equivalence classes of field nodes can partition the field based on influences.

4.3.3 Influence Graph Properties

Why is an influence graph useful? As Figures 4.13 and 4.14 showed for thermal hills, influence strengths vary in different directions and from different locations, depending on geometry and field properties. Influence graphs encapsulate these variations in dependence for use by other reasoning mechanisms. For example, control
- \textit{evaluate} : (\text{Space} c, \text{Space} f, \text{Influence}\_\text{Graph}\_\text{Constructor} n) \\
\quad \rightarrow (\text{Influence}\_\text{Graph} g)

- Constructs an influence graph storing influences from the controls on the field nodes.
- \( g = (f, c, c \times f, w) \) where \( w \) is computed in an implementation-specific manner according to \( n \).

- \textit{influence\_from} : (\text{Influence}\_\text{Graph} g, \text{Spatial}\_\text{Object} c, \text{Space} s) \rightarrow (\text{Field} i)

- Returns a field encapsulating the influences from the control on the nodes.
- \( i : s \rightarrow \mathcal{R} \) such that \( i(f) = w((c, f)) \) for \( f \in s \).

- \textit{influence\_to} : (\text{Influence}\_\text{Graph} g, \text{Spatial}\_\text{Object} f, \text{Space} s) \rightarrow (\text{Field} i)

- Returns a field encapsulating the influences on the node from the controls.
- \( i : s \rightarrow \mathcal{R} \) such that \( i(c) = w((c, f)) \) for \( c \in s \).

Table 4.8: Primary influence graph operations.

Placement design will exploit the constraints on heat flow indicated by directions of flow.

Influence graphs also encapsulate locality of control effects. Locality can be used to distinguish between a \textit{near field} and \textit{far field} relative to the amount of influence exerted by a control. For example, Figure 4.15 shows the near and far fields based on the thermal hill from a heat source.

\textbf{Definition 12 (Near and Far Field)} \textit{The near field of a control is the set of nodes strongly influenced by the control; the far field is the set of nodes weakly influenced by the control.}
Figure 4.15: An influence hill partitions a field into near and far fields relative to a control.

| • is\_near : (Influence\_Graph g, Spatial\_Object c, Spatial\_Object f) → (Boolean b) |
| Indicates whether or not the field node is in the near field of the control node. |

Table 4.9: SAL near-field/far-field test operator.

The new SAL operator `is\_near` (Table 4.9) performs this partitioning based on user-specified criteria regarding the weights in an influence graph. Possible implementations, easily layered over the SAL classify operator, use a fixed influence threshold, a threshold proportional to the peak, or a threshold based on the "knee" of the hill. Control parameter design will leverage the locality encapsulated in influence graphs to support more independent reasoning about control actions taken by decentralized controls.

In many distributed physical phenomena, despite nonlinearities in the spatial variables (e.g. non-uniform conduction characteristics or irregular geometry), solutions can be linearly superposed.
Definition 13 (Linear Superposability in the Heat Equation) Heat equation solutions are linear superposable: if $\phi_1$ and $\phi_2$ are solutions to the heat equation and $c$ is a constant, then

- (Scalability): $c\phi$ is a solution to the heat equation.

- (Additivity): $\phi_1 + \phi_2$ is a solution to the heat equation.

Boundary conditions impact the solution to a heat equation — it would not be correct to “double count” boundary conditions when adding two solutions. If the boundary conditions specify a non-zero constant value (i.e. the temperature in the domain has little impact on the temperature outside, which remains constant), the impact can be treated similarly to a heat source/sink, and added in as a separate part of the solution. If the boundary conditions have a term involving $\phi$ (e.g. an insulated domain with gradient boundary conditions), the impact is factored directly into the capacitance matrix for the system.

Since influence graphs represent solutions to the heat equation at unit control values, this property means that the effects of controls can be combined through a superposition of influences. For example (see Figure 4.16), given the thermal hill for a heat source at one control value, the temperature field resulting from a different control value is simply an appropriately scaled version of the original thermal hill. Similarly, given the thermal hills for two separate heat sources, the temperature field resulting from both heat sources is simply the sum of the two thermal hills. Influence graphs encode the crucial dependency information, while hiding other possibly nonlinear effects. Control parameter design will exploit linear superposability to efficiently evaluate fields through addition and subtraction of appropriately scaled hills.
4.3.4 Alternate Views of Influences

A common representation of the structure of a field is with iso-contours, or curves of equal field value. For example, Figure 4.17 shows some iso-contours for the hill of Figure 4.14. The contours are essentially loops around the hill at the same "altitude" (influence value). The steepness of the hill can be judged by examining the distance between adjacent contours at various points. This in turn indicates the rate of influence decay in different directions.

Gradient vectors are a dual representation to iso-contours: while iso-contours loop around a hill, gradient vectors point up it. For example, Figure 4.18 shows the gradient vector directions for the hill in Figure 4.14. The lengths of these vectors, not shown here, indicate the local steepness of the hill. The directions of the vectors indicate the directions of steepest ascent up the hill.
Figure 4.17: Iso-influence contours for material and heat source.

Figure 4.18: Gradient vector directions for material and heat source.
SAL field operators support local estimation of gradient vectors by approximating the spatial derivatives.

Gradient vectors can be bundled into trajectories (e.g. Figure 4.19), following the approach of Section 3.4.3. These gradient trajectories show the main paths of influence flow through the field landscape.

4.4 Control Placement Design

The first design task considered in this chapter is that of designing a control placement. For the thermal domain, control placement design uses a description of a material’s geometry, conduction properties, and boundary conditions, in order to place heat sources.
Definition 14 (Control Placement Design) Control placement design yields a number \( n \) and set of control locations \( C = \{c_1, c_2, \ldots, c_n\} \) satisfying a specified design objective.

The placement of the heat sources affects their ability to achieve a desired temperature distribution by parametric adjustment. For example, if all the heat sources are clumped at one end of the material, they cannot adequately control the area at the other end of the material. Similarly, such a clumping makes it hard to individually determine control actions, since the actions taken by one control strongly affect the necessary actions of another (e.g. heat from one source affects the area another source is trying to control). Furthermore, as discussed in the introduction, in order to scale up to massive sets of distributed controls, it is necessary to reason about controls as independently as possible.

Based on this insight, the design objective considered here is that of placing controls so that they minimally interfere with each other. The approach to achieving this objective is to decompose a problem domain into a set of decoupled, atomic subregions, and then independently design controls (placement and parameters) for the separate subregions. Regions are considered decoupled if the exact control design in one region is fairly independent of the exact control design in another. A region is considered atomic if it needs no further decomposition — control design for the region yields adequate control of the region. Influence graphs are used to perform this decomposition; refer again to Figure 4.5 in Section 4.1 for an overview.
4.4.1 Control Probes

For a temperature field to exhibit structures, heat sources must be applied; then an influence graph can be constructed. For example, Figure 4.20 shows the iso-influences resulting from two different heat source placements; in both cases, the structure of the contours indicates the constraint on heat flow due to the narrow channel. The control placement design algorithms in the following subsections are based on the response of temperature fields to such control probes.

**Definition 15 (Control Probe)** A control probe is a sample control placed in a domain in order to estimate effects of other potential controls.

The number and placement of control probes affects the structures uncovered in temperature fields, and thus the quality of the resulting control design. Probe locations can be chosen either statically or dynamically. For example, static probe locations can be chosen at random or based on the size of the field discretization (e.g. every 10 units). Dynamic probe locations can be chosen in order to gather more information about inadequately explored regions or to disambiguate inconsistent interpretations. This allows potentially better results at the expense of more implementation complexity and run-time cost.
Probes serve as representatives for the effects of arbitrarily-placed controls. In particular, the probes that most strongly affect a location serve as the basis for an approximation of the effects that would be produced by a control placed at that location.

**Definition 16 (Primary Controls)** The primary controls for a location are those controls that most affect the location, according to a user-specified definition of "most affect."

For example, the primary controls for a node could be those controls with an influence greater than some threshold, those controls for which the node is in the near field, or those controls whose influence is a standard deviation above the average influence from all controls. Note that nodes can have multiple primary controls.

The quality of the approximation of controls at arbitrary locations by representative primary controls depends on geometry and material properties. Since the influence graph encapsulates the effects of geometry and material conditions, it provides a natural mechanism for reasoning about approximation quality. In particular, experimental results presented later in this section show the trade-off between number of probes (and thus approximation quality, assuming that the quality of an approximation for a location improves with more, closer probes) and resulting quality of control design.

By taking control probes as representatives of control placement effects on a field, the problem of decomposing the domain into regions can be reformulated into one of partitioning probes into equivalence classes. Each equivalence class of probes serves as a representative for the effects of its *controlled region*, the set of nodes for which
the probes are primary controls. A good decomposition produces probe classes whose controlled regions are decoupled from the controlled regions of other classes, and which have no acceptable subclasses.

New SAL operators, detailed in Table 4.10, support the probe process. The \textit{probe} operator provides a common interface to the task of choosing probe locations and computing the resulting influence graph. The \textit{primary_controls} operator defines which controls have the strongest influences on a set of nodes. Finally, the \textit{controlled_region} operator inverts the map defined by \textit{primary_controls}, defining for which nodes a set of controls are the strongest influences.

\subsection{4.4.2 Evaluating Control Decoupling}

The first criterion for evaluating a decomposition is that each region be decoupled from other regions.

\textbf{Definition 17 (Decoupling)} \textit{Two regions are decoupled if controls in one region have little effect on nodes in the other, and vice-versa.}
Figure 4.21: Similarity of flows in the far fields of control probes suggests indistinguishability of control placement.

In terms of control probe equivalence classes, decoupling will be evaluated by considering independence of control placement and the independence of control parameters.

To evaluate independence of control placement, consider the influence gradient vectors induced by a set of probes; Figure 4.21 shows a simple example for two probes. While the flows are different near the locations of the two probes, they are quite similar in direction far away from the probe locations. This similarity is due to constraints imposed by geometry and material properties; in this case, the narrow channel of the material effectively decouples the left and right halves. A numerical measure for the similarity is easily implemented in SAL; for example, by averaging over the far field the angular difference between gradient vectors produced by different probes. This measure evaluates the indistinguishability of exact control placement within the set of probe locations, and thus is correlated with a good decomposition into decoupled regions.
influence\_similarity : (Influence\_Graph g, Space s) \rightarrow (Number n)
Measures the similarity in influence of the set of controls.

Table 4.11: SAL influence similarity operator.

To evaluate independence of control parameters, recall the distinction between a probe's near field and its far field: the far field is only weakly influenced by the probe, and thus can be effectively decomposed from it. Alternatively, it makes sense to group together probes that have significant overlap in their near fields. This overlap can be measured in SAL by element-wise comparing influence value differences and summing the results.

The new SAL operator influence\_similarity (Table 4.11) provides a common interface for these and other metrics of similarity. Such metrics provide the basis for grouping probes into equivalence classes of similar probes, discussed later in this section.

In addition to independence from any single control, a well-decoupled region must be independent from the combined effects of other controls. That is, for a set of probes, the total influence on its controlled region from controls for other probe sets must be sufficiently small.

4.4.3 Evaluating Region Atomicity

The second criterion for evaluating a decomposition is that each region be decomposed far enough. For example, in Figure 4.22 a partition \{\{A, B, C, D\}, \{E, F, G\}\} achieves good decoupling, since the probes in the first class are relatively independent
from those in the second class. However, it is not atomic, since \( \{A, B, C, D\} \) can be further decomposed into \( \{\{A, B\}, \{C, D\}\} \).

**Definition 18 (Atomic Decomposition)** A decomposition region is atomic if none of its subregions are adequately decoupled.

One approach to ensuring atomicity of the classes of a decomposition is to recursively test subsets of probes to see if they result in valid decompositions. For example, by testing partitions of the class \( \{A, B, C, D\} \) for independence, the partition \( \{\{A, B\}, \{C, D\}\} \) would be uncovered. The test can use heuristics to avoid testing all possible subclasses. For example, just by examining overlap in influences in the class \( \{A, B, C, D\} \), the partition \( \{\{A, B\}, \{C, D\}\} \) can be generated as a counterexample to the atomicity of \( \{A, B, C, D\} \). If a class is already small, out-of-class probes can be used in such a test, and, if necessary, new probes can be introduced. For example, in an atomicity test for \( \{A, B\} \), checking independence of \( \{A, C\} \) from \( \{B, D\} \) would show that \( \{A, B\} \) is indeed atomic. A cheaper and empirically effective method is to allow grouping of pairs of probes only if their near fields sufficiently overlap, as shown in Figure 4.23. The SAL operator *atomicity* (Table 4.12) provides a high-level interface to these metrics.
4.4.4 Control Probe Partitioning

Based on these criteria, control probes can be clustered into decoupled, atomic equivalence classes. One effective method for performing this clustering is to use the SAL neighborhood graph and classifier mechanisms. Start with each probe in its own class, and form a neighborhood graph of classes based on proximity (e.g., Delaunay triangulation or nearness neighborhood). Then greedily merge neighboring pairs of classes that are most similar, as long as a region is strongly influenced by other regions, and until a merger would result in a non-atomic class. This is implemented in the SAL classifier mechanism by combining a merging cluster algorithm with a best-match equivalence predicate. Table 4.13 provides pseudocode for this algorithm, and Figure 4.24 shows the data flow. Figure 4.25 illustrates a sample probe neighborhood graph, Figure 4.26 depicts some influence gradients for sample probes, and
While classes are *atomic*:

For each class:

- Measure *influence similarity* with *neighbor* classes.
- *Merge* best-match pairs of classes.

Table 4.13: SAL algorithmic description of probe clustering.

Figure 4.27 shows the controlled regions for equivalence classes of probes after the merging process.

### 4.4.5 Performance

The influence-based decomposition algorithm has proved effective in designing control placements for decentralized thermal regulation. The performance has been measured in two ways: quality of the decomposition, and ability of the resulting control design to achieve an objective.

One important question about the design algorithm is the impact of the number of control probes on the effectiveness of the resulting design. To test this, different numbers of probes (4, 8, 16, and 32) were placed at random in a given domain, and results were averaged over a number of trial runs. While smarter probe placement techniques might yield more consistently effective designs, this approach provides a baseline and illustrates the trade-off between number of probes and error/variance.

Data for three sample problems are given here: a plus-shaped piece of material, a P-shaped piece of material, and an anisotropic (non-uniform conduction coefficient) bar. These problems illustrate different geometries, topologies (the P-shaped material...
Figure 4.24: SAL data flow for probe clustering algorithms: repeatedly merge best-match pairs of classes.

Figure 4.25: Probe merging example: probe neighborhood graph.
Figure 4.26: Probe merging example: influence gradient vectors from two probes.

Figure 4.27: Probe merging example: region decomposition after merging.
has a hole), and material properties. Other problems have also been tested: the results are similar.

The simple merging algorithm outlined above was used to decompose the input domain. The near field of each probe was set to all nodes with influence at least 10 percent of the peak. The probe neighborhood graph was a Delaunay triangulation. Similarity measures between probe classes compared flow vector direction differences. Merging was performed until four classes remained.

**Decomposition Quality**

The goal of the decomposition algorithm is to partition a domain into regions such that source placement and parametric optimization in each region is relatively independent of that in other regions (decomposed) and has no internally independent regions (atomic). The estimate of the quality of a decomposition used here is based on a corresponding formalization for image segmentation by Shi and Malik [70]. Compare the total influence from each control location on locations in other regions (decomposed), and the amount of influence from that location on locations in its own region (atomic). To be more specific, define the decomposition quality \( q \) (\( 0 \leq q \leq 1 \)) for a partition \( P \) of a set of nodes \( S \) as follows (\( i \) is the influence):

\[
q = \prod_{R \in P} \sum_{c \in R} \frac{\sum_{r \in R} i(c, r)}{\sum_{s \in S} i(c, s)}
\]

For each control node, divide its influence on nodes in its own region by its total influence. Summing that over each region yields an estimate of the fraction of control output of any control location in the region that is used to control the other locations in that region. The quality measure is combined over all regions by taking the product of each region's quality.
<table>
<thead>
<tr>
<th></th>
<th>Spectral</th>
<th>SAL4</th>
<th>SAL8</th>
<th>SAL16</th>
<th>SAL32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>0.97</td>
<td>1.06</td>
<td>1.21</td>
<td>1.17</td>
</tr>
<tr>
<td>std dev</td>
<td>n/a</td>
<td>0.30</td>
<td>0.17</td>
<td>0.12</td>
<td>0.10</td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>0.68</td>
<td>0.77</td>
<td>0.86</td>
<td>1.11</td>
</tr>
<tr>
<td>std dev</td>
<td>n/a</td>
<td>0.063</td>
<td>0.15</td>
<td>0.12</td>
<td>0.066</td>
</tr>
<tr>
<td>Bar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>0.56</td>
<td>0.65</td>
<td>0.76</td>
<td>0.85</td>
</tr>
<tr>
<td>std dev</td>
<td>n/a</td>
<td>0.26</td>
<td>0.10</td>
<td>0.073</td>
<td>0.024</td>
</tr>
</tbody>
</table>

Table 4.14: Performance data for decomposition quality: relative average and standard deviation of decomposition quality. Spectral uses spectral partitioning to decompose a full influence graph, while SAL4-SAL32 use the probe-based method with 4-32 randomly-placed probes.

Shi and Malik [70] applied spectral partitioning (refer again to Section 4.2) to a matrix equivalent to an influence graph with edges weights modulated by total influence from corresponding control nodes. Intuitively, this approach partitions the influence graph, removing edges so that the resulting connected regions maximize internal influence (atomic) and minimize external influence (decomposed). Shi and Malik showed that this approach yields a good estimate of the optimal decomposition, so it is used here as a baseline for the performance data.

Table 4.14 provides raw performance data: the average and standard deviation of the decomposition quality over the set of trial runs. Figure 4.28 illustrates the variation in error and standard deviation with respect to different numbers of control probes.

For all three problems in Figure 4.28, the average quality naturally decreases as the number of probes decreases. (There is a slight taper in the performance for
Figure 4.28: Performance data indicate that the decomposition algorithm supports trading decomposition quality for computation. Decompositions achieve quality comparable to spectral partitioning, but with an influence graph for a small number of probes rather than a full influence graph.
the plus shape, due to statistical sampling.) Furthermore, the standard deviation of quality tends to increase as the number of probes decreases, since the partition is more sensitive to specific probe placements. The curve indicates a trade-off between the amount of computation versus the resulting decomposition quality. With enough probes, the decomposition quality is roughly equivalent to that of spectral partitioning. It is worth noting that spectral partitioning requires computation of a matrix corresponding to a full influence graph (from every node to every other node), rather than just influence from a small number of probes. The spectral partitioning approach also requires solving a general eigenvalue problem for the influence matrix.

Control Placement Quality

The ultimate measure of the control design algorithm is how well a design based on a decomposition can achieve a control objective. This section evaluates the ability of decomposition-based control designs to achieve a uniform temperature profile. This profile is better than other, non-uniform profiles at indicating the performance of a decomposition, since it does not depend as much on local placement adjustment and parametric optimization. Intuitively, if a decomposition clumps together sources, then some other region will not get enough heat and thus will have a large error.

Simulated annealing [54] serves as a baseline comparison for error: the annealing process was run for 100 steps (equivalent computation to placing 100 probes). The decomposition-based control design used a simple approach: for each region of a decomposition, place controls in the “center of influence” (like the center of mass, but weighted with total influence from the probes, rather than mass, at each point). In both cases, only the global control placement was designed; local placement adjustments could somewhat reduce the error.
Table 4.15 provides the raw performance data, including the average error (sum of squared difference between actual temperature profile and desired temperature profile) and the standard deviation in the error over the set of trial runs. Figure 4.29 illustrates the variation in error and standard deviation with respect to different numbers of control probes.

As with decomposition quality, the average and standard deviation of control quality tend to improve with the number of probes. With enough probes, the quality is commensurate with that of simulated annealing. A major difference is that the decomposition-based approach uses a small, fixed number of function evaluations.

### 4.4.6 Discussion

The control placement design algorithm decomposes a domain based on influence graph structures for a set of control probes. It uses influence graphs to measure similarity of effects on the field from classes of probes. Based on these measures, it

<table>
<thead>
<tr>
<th></th>
<th>Anneal</th>
<th>SAL4</th>
<th>SAL8</th>
<th>SAL16</th>
<th>SAL32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Plus</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>1.25</td>
<td>1.07</td>
<td>1.03</td>
<td>1.0</td>
</tr>
<tr>
<td>std dev</td>
<td>0.014</td>
<td>0.083</td>
<td>0.069</td>
<td>0.028</td>
<td>0.029</td>
</tr>
<tr>
<td><strong>P</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>1.16</td>
<td>1.17</td>
<td>1.03</td>
<td>0.99</td>
</tr>
<tr>
<td>std dev</td>
<td>0.014</td>
<td>0.077</td>
<td>0.124</td>
<td>0.014</td>
<td>0.027</td>
</tr>
<tr>
<td><strong>Bar</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>1.0</td>
<td>1.42</td>
<td>1.11</td>
<td>1.0</td>
<td>0.99</td>
</tr>
<tr>
<td>std dev</td>
<td>0.011</td>
<td>0.24</td>
<td>0.13</td>
<td>0.085</td>
<td>0.050</td>
</tr>
</tbody>
</table>

Table 4.15: Performance data for decomposition-based control design: relative average and standard deviation of error. Anneal is a simulated-annealing based optimizer, while SAL4–SAL32 use the probe-based method with 4–32 randomly-placed probes.
Figure 4.29: Performance data indicate that decomposition-based control placement design supports trading control quality for computation. Designs achieve performance comparable to simulated annealing, but with a small, fixed number of field evaluations.
identifies minimal equivalence classes of probes that are mutually independent with respect to control placement and parameter design.

In order to provide a consistent basis for comparison with other algorithms, the performance data presented here applied probe merging until a desired number of regions was reached. However, the general framework actually supports computation of the desired number of control nodes, based on probe class atomicity.

Section 4.2 presented spectral partitioning as a technique for decomposing a field in order to solve the heat equation on separate subfields. Spectral partitioning and other graph partitioning algorithms [44, 71, 55, 35] could be used directly for control design: partition a field into subfields and place a control in each subfield. However, this approach accounts for topology and perhaps geometry, but not material properties. This section presented spectral partitioning on the influence graph (rather than on the domain) as a baseline for decomposition performance data. As previously mentioned, the SAL approach is much more efficient, using influences for only a small number of probes, rather than for every node. The SAL partition process could be implemented by encoding probe similarity metrics in graph edges and then applying a graph partitioning algorithm. However, the merging algorithm presented here is simple, efficient, and effective.

The SAL design algorithm searches the design space in a much different manner from that of other combinatorial optimization algorithms, such as genetic algorithms [36] and simulated annealing [54]. Rather than (perhaps implicitly) searching the space of all possible combinations of source locations, SAL combines results from a small set of control probes, develops a global description of the domain, and partitions it appropriately. SAL explicitly forms equivalence classes and structures in the
domain, rather than implicitly representing them in terms of, for example, increased membership of highly-fit members in a population. Since SAL design decisions are based on the influence structure of the field, this approach supports higher-level reasoning about and explanation of its results; for example, a design decision could be explained in terms of constrained influence flows through a field.

Both decomposition quality and control quality increase somewhat asymptotically with the number of probes. The major computational cost is in computing probe influences, rather than in merging probe classes. This suggests a modified control design algorithm that iteratively increases the number of probes, checks the resulting decomposition at each step, and halts when the quality stabilizes. This algorithm avoids dependence on a fixed number of probes and follows a trade-off curve between computation and control quality.

4.5 Control Parameter Design

Given a control placement design, the next task is to optimize control parameters in order to satisfy the design objectives. Consider the control objective of maintaining a specified temperature distribution for an extended period of time, as is the case for rapid thermal prototyping [43]. This task can be broken into two parts: design-time computation of set points around which the heat source outputs will vary, and run-time feedback control of the actual heat source outputs based on local, linearized models derived for those set points. This section considers the computation of the set-point heat source values, leaving the feedback control to standard engineering techniques (e.g. as in [43] or [21]).
Repeat
   For each control node
     For each control action adjustment
       Compute the resulting error.
     Choose the best adjustment.

Table 4.16: SAL algorithmic description of decentralized optimization algorithm.

Definition 19 (Control Parameter Design) Control parameter design for a set of controls $C = \{c_1, c_2, \ldots, c_n\}$ and a set of control parameter spaces $\{P_1, P_2, \ldots, P_m\}$ yields mappings $f_1: C \rightarrow P_1$, $f_2: C \rightarrow P_2$, $\ldots$, $f_m: C \rightarrow P_m$.

Control parameter design for a distributed thermal field determines output values for a set of heat source positions. This requires simultaneous optimization of many parameters (the heat source values). While algorithms for multi-parameter optimization exist [61], they are computationally expensive for large problems and difficult to parallelize for distributed applications. This section demonstrates that structural knowledge, in the form of the influence graph, significantly improves the performance of a basic decentralized optimization algorithm.

A simple decentralized optimization algorithm repeatedly tests adjustments to control output and chooses the one that minimizes the error (e.g. sum of squared difference of the resulting temperature from the desired profile). Table 4.16 and Figure 4.30 summarize the algorithm and its data flow. Remember that the optimization processes are decentralized, so that each heat source adjusts itself independently, taking a step towards what it thinks minimizes error.
Figure 4.30: SAL data flow for the basic decentralized optimization algorithm: adjust control values based on the error in the field resulting from different heat outputs.

\begin{table}[h]
\centering
\begin{tabular}{|c|}
\hline
\texttt{optimize : (Space s.} \\
\texttt{Set of (Spatial\_Object \rightarrow \textbf{()}) a,} \\
\texttt{(Spatial\_Object \rightarrow Number) e) \rightarrow \textbf{()}} \\
\texttt{Repeatedly adjusts each control with the modifications a, choosing the adjustment that produces the best error, measured by e.} \\
\hline
\end{tabular}
\caption{SAL decentralized optimization operator.}
\end{table}

This decentralized optimization process is generalized and encapsulated in the new SAL operator \texttt{optimize}, detailed in Table 4.17. Different implementations could choose to synchronize control updates or to anneal the adjustments.

In the next three sections, the SAL influence graph mechanism will be used (1) to avoid redundant computation during field evaluation, (2) to reduce communication among sources and field nodes, and (3) to support cooperation among local optimization processes for the sources.
For each control node \( c \)
   For each field node \( f \)
       Update the temperature by the influence of \( c \) on \( f \),
       scaled by the control adjustment for \( c \).

Table 4.18: SAL algorithmic description of efficient field evaluation.

### 4.5.1 Efficient Field Evaluation

During each step of an iterative optimization process, the field is evaluated using the relatively expensive, iterative relaxation method on the spatial objects, as discussed in Section 4.2. However, recall that an influence graph caches the dependence of field nodes on normalized sources, and that the field is determined by a linear superposition of source effects. Thus the field value for a node can be calculated by summing together the weights of influence graph edges coming into the node, scaled by the control source values. This computation is extremely fast and results in a drastic speed-up in computation.

Table 4.18 summarizes the field-evaluation algorithm, and Figure 4.31 illustrates the data flow for the modified optimization algorithm. To determine the impact of a different heat output, a source calculates the resulting temperature change for each field node, based on influence graph edge weights.

Table 4.19 describes the new SAL operator `update_field` that abstracts this process for general influence-based field updates. This operator could serve as part of the error evaluation function for the `optimize` operator.
Figure 4.31: SAL data flow for efficient field evaluation: adjust temperatures based on changes in source values and influence graph information.

\[ \text{update\_field : (Spatial\_Object c, Influence\_Graph g, Field f, Number s)} \rightarrow () \]

Updates the field by adding the scaled influence for the control.

Table 4.19: SAL influence-based field update operator.
The influence graph essentially pre-computes and caches the inverse of the capacitance matrix of the field. An important distinction is that it does this in a decentralized fashion, without ever forming a global matrix for the temperature field or the sources. This representation is particularly efficient when sources are sparse.

4.5.2 Reduced Communication

At each optimization step, a source must estimate the error caused by an adjustment to the source value, with respect to the current state of the temperature field. The source can consult the entire temperature field for the current error, and then adjust the values throughout the field when it changes, but that requires much communication. Alternatively, it can consider only a local region assigned to it (e.g. the region for which it was designed in Section 4.4), but that ignores the influence on the other regions. Better yet, a source can communicate more frequently with those field nodes it most strongly affects, as shown in Figure 4.32. If a source only weakly affects a temperature node, we need not assign it much blame/credit for the error at that node.

Table 4.20 summarizes the new field-evaluation algorithm, and Figure 4.33 illustrates the new data flow during control optimization. The frequency of source-field communication is a function of the amount of influence. Decreasing frequency decreases overall communication costs, but increases the potential for error due to underestimated source effects.

There are several possible strategies for establishing source node to field node communication. The most basic method computes communication frequency as a function
Figure 4.32: Influence structure supports reduced communication between control node and field nodes.

For each control node $c$
  For each field node $f$
    Every $\text{freq}(\text{infl}(c, f))$ iterations
      Update the temperature by the influence of $c$ on $f$, scaled by the control adjustment for $c$.

Table 4.20: SAL algorithmic description of reduced-communication field evaluation.

Figure 4.33: SAL data flow for reduced communication optimization: modulate frequency of source-field communication by influence strength.
of the weight along the influence graph edge. This requires each source to communicate with each field node (some more frequently than others). A more qualitative method forms equivalence classes of field nodes based on influence (iso-influences) for each source, and treats the regions equivalently with respect to communication frequency. Now communication paths only exist between sources and regions. An even more qualitative method forms equivalence classes of field nodes based on which source has the strongest influence, again treating regions equivalently with respect to communication frequency. With this assignment, each source communicates only with its own region and with other sources, which pass information on to their regions. These different strategies serve as different implementations of the $update_field$ operator of Table 4.19, and are incorporated into the decentralized control algorithm as part of the error evaluation function.

4.5.3 Joint Optimization

While the decentralized optimization algorithm seeks to independently optimize sources, in reality there is coupling: the heat from one source affects the temperature throughout the entire field and thus influences the actions taken by other sources (refer again to Figures 4.13(b) and (c)). Independent optimization of coupled sources might require more iterations to converge, as the sources make seemingly independent choices which they later find to be wrong due to dependencies. Even worse, sources might converge to sub-optimal values, where no independent actions help, but cooperative actions would.

As a particular example of cooperative optimization, consider the "ridge problem" faced by optimization techniques. An example manifestation of the ridge problem in
Figure 4.34: Ridge problem in control optimization: independently increasing or decreasing either control output increases the error, but jointly decreasing control 1 and increasing control 2 decreases the error.

the temperature control domain, illustrated in Figure 4.34, occurs when independently increasing the value of one source increases the total error and independently decreasing the value of another source also increases the total error, but jointly increasing the one and decreasing the other decreases the total error. This is due to coupling between the areas influenced by the sources: the joint modification maintains a similar temperature profile in the overlap area and benefits other areas. By cooperatively optimizing, the optimizer walks along the ridge in the error landscape.

Joint optimization can be programmed in SAL by incorporating supervisors into the decentralized optimization algorithm. A supervisor is a control node whose action is to shift control output from one control to another, as in Figure 4.35. Supervisors can be placed, for example, between pairs of very close controls, or between pairs of controls whose influence hills are highly overlapping. Since a supervisor's action
Supervisors implement source cooperation and help avoid optimization ridges by shifting heat from one source to the other based on the error profile in the field. This approach could be extended to the addition of supervisor controls that shift heat among groups of sources rather than between pairs. Note that supervisors only need to be established between pairs of sources that are tightly coupled. Further extensions could let supervisors check for cooperation less frequently or recognize a potential need for cooperation (for example, too much heat near one source and not enough near the other) before thoroughly testing.
Place supervisors based on proximity, influence overlap, etc.

Supervisor adjustment function:
Adjust control value for supervised controls.

Supervisor error function:
Update field with influence difference of supervised controls.

Add supervisors to set of decentralized controls for optimize.

Table 4.22: SAL algorithmic description of supervisor optimization.

Figure 4.36: SAL data flow for joint optimization: form supervisor nodes for tightly-coupled sources; optimize supervisors, shifting heat from one source to another based on errors in the field.
4.5.4 Performance

The control parameter design algorithms, when applied to several problems, result in competitive designs and run-time performance.

For a distributed optimization problem with $M$ sources and $N$ field objects, the basic algorithm requires on the order of $kLMN$ units of computation, where $k$ and $l$ are the numbers of iterations for the optimization and relaxation processes respectively. $k$ and $l$ depend on properties of the problem including the size of the field, the number of controls, the material properties, and the geometry; in the test cases below, $k$ is roughly between 10 and 100, while $l$ is roughly between 100 and 1000. Using the influence graph to eliminate repeated relaxation, the algorithm scales as $kMN$. Exploiting the communication structure, the cost is reduced to $kMNC_N$ for a smaller $C_N$, the number of field objects with which each source communicates, possibly independent of $N$. By cooperating among the local optimizers, the number of iterations $k$ is further reduced.

Efficient Field Evaluation

As expected, the influence graph mechanism results in enormous savings during repeated decentralized field evaluations. For example, in an implementation using the C++-based SAL library on a 100-MHz Pentium system with Linux and gcc, it takes about 49 seconds to iteratively solve for the temperature in a field with about 1000 nodes, while it takes less than 0.02 seconds using the influence graph. Since the field evaluation must be performed at each iteration, the savings add up quickly.
Reduced Communication

Influence graphs significantly reduce communication during source optimization. Table 4.23 summarizes results for steady-state parametric design on a regular 20x20 discretized thermal field. While the domain evaluated here is square, similar results hold for other shapes — the important factor is the locality of the thermal hills encapsulated in the influence graph. Data for three problems are provided: four sources near the corners of the grid, four sources near the center of the grid, and sixteen sources tiled over the grid. These three problems exhibit varying thermal hill shapes and thus varying ability to reduce communication. Three performance results are shown for each test: the number of iterations for convergence, the total source-field node communication, and the average squared error across the thermal field. Actual run-time is roughly proportional to the number of communications.

The first two optimizers evaluated (Gauss-Newton and Broyden-Fletcher-Goldfarb-Shanno) are Matlab-based implementations of two standard multi-parameter optimization algorithms (Simplex search optimization is not included because it fails to converge within 300 steps on all of these tests.) Note that the Matlab algorithms are not decentralized; in order to compare the amount of communication, each source is considered to communicate with each non-boundary field node each iteration. The SAL-based optimizers use an implementation with varying amounts of communication: SAL1 updates each field object based on each source every iteration, while SAL2-SAL4 update field objects with frequency proportional to influence, with different constants of proportionality. Performance numbers are relative to Gauss-Newton (lower is better).
Table 4.23: Performance data for communication reduction in optimization: relative number of iterations, number of communications, and resulting error for different optimization methods for representative problems. GN (Gauss-Newton) and BFGS (Broyden-Fletcher-Golfarb-Shanno) are Matlab-based centralized optimizers. SAL1-SAL4 are influence graph-based decentralized optimizers with communication rates proportional to influence, with varying constants of proportionality. Values are relative to Gauss-Newton.
These results show that on representative multi-parameter optimization problems, the SAL structure-based decentralized optimizers compete well with the centralized optimization techniques in both speed and error, while greatly reducing the amount of communication among distributed optimization processes. Figure 4.37 charts the trade-off between communication and error in the four SAL optimizers on these problems. Naturally, error increases as communication decreases, but there is quite a long flat area where the communication decreases without a serious impact on the error. In problems with larger domains, there will be even fewer field nodes strongly influenced by a source (depending of course on geometry and material properties). providing even greater potential savings.

Joint Optimization

Influence graphs also support cooperative source optimization. Table 4.24 provides data for representative problems with tight coupling among sources due to material properties and source spacing. The sources are placed in four different configurations on a 20x20 grid: a pair of sources at the edge, four sources tightly packed near a corner, eight sources in a line across the middle, and sixteen sources tightly packed near the center. The results from the two (centralized) Matlab optimizers are provided for reference; the first SAL optimizer does not cooperatively optimize, while the second one places a supervisor between each neighboring pair of sources. BFGS fails to converge for the 8-line test case.

Figure 4.38 illustrates the convergence rate of the different algorithms. Both SAL optimizers find or come very close to the optimal error, but the use of cooperation generally results in much faster convergence. In the final test case (16 sources tightly packed), the cooperative optimization method takes somewhat longer. This is most
Figure 4.37: Performance data indicate that influence graphs support trading optimization quality for amount of communication. In the flat area, amount of communication is greatly reduced with little impact on error.
Table 4.24: Performance data for cooperative optimization: number of iterations and resulting error for different optimization methods for representative problems. GN (Gauss-Newton) and BFGS (Broyden-Fletcher-Golfarb-Shanno) are Matlab-based centralized optimizers. SAL is the standard SAL decentralized optimizer, while SAL-coop uses influence graph-based joint optimization. Values are relative to Gauss-Newton.
Figure 4.38: Performance data indicate that influence graphs support cooperative optimization: SAL-coop uses supervisors for pairs of tightly-coupled sources and generally requires fewer iterations than does the standard SAL optimizer. The centralized Matlab optimizers GN (Gauss-Newton) and BFGS (Broyden-Fletcher-Golfsb-Shanno) are provided for reference.
likely due to the implementation of only pairwise cooperation — the tight coupling of so many sources might benefit from hierarchical supervision of larger groups of sources.

4.5.5 Discussion

Influence graphs support control parameter design by encoding structural dependencies among control sources and spatial fields. This information allows efficient evaluation of fields in terms of scaled sums of influences. It also supports trading off among computation, communication, and control quality based on amount of influence. While the SAL optimization algorithm was based on a very simple decentralized updating process, its results are competitive with standard centralized optimization algorithms.

4.6 Summary

This chapter has applied Spatial Aggregation Language data types and operators to modeling, interpreting, and controlling distributed physical systems. SAL data types provide a natural framework for programming these tasks, and high-level operators abstract common computational patterns.

Section 4.2 discussed SAL encodings of traditional science and engineering routines for partial differential equations. Figure 4.39 summarizes the data flow among components and the high-level SAL operators. These components and operators abstract powerful problem-solving techniques, including discretization, relaxation, multi-level solving, and problem decomposition, which are applicable to a wide variety of partial differential equations. The SAL encodings emphasize local interaction rules at multiple levels of abstraction.
Figure 4.39: SAL operators support high-level programs elaborating the consequences of partial differential equations.
The SAL local-interaction framework is a powerful and convenient programming mechanism, and it mirrors physical intuition about these processes. However, traditional mathematical techniques have other advantages. Mathematical formalisms allow proofs of convergence and numerical stability properties of partial differential equation solvers in terms of matrix properties such as spectral radius and condition number. In some cases, solving through local interaction rules might not be as efficient as applying other techniques (e.g. inverting a matrix). Finally, interacting with other pieces of mathematical software often requires adopting the global matrix viewpoint.

Section 4.3 introduced the influence graph mechanism encoding effects of decentralized controls on distributed physical fields, and Sections 4.4 and 4.5 applied it to control placement and control parameter design, respectively. While this chapter has concentrated on the specific application of decentralized control of heat, many of these techniques generalize to other, similar application areas. The SAL influence graph operators developed in this chapter provide a high-level framework, illustrated in Figure 4.40, for decentralized control design. Probing a field allows extraction of a structural representation (the influence graph) of the effects of controls on the field. Control probes are classified based on similarity and atomicity measures, and the controlled regions of the equivalence classes yield a decomposition of the original field. Actual controls are placed based on this decomposition. Parametric optimization adjusts the control actions of these controls, based on their effects on the field as encapsulated in an influence graph.
This generic framework is applicable to a variety of decentralized control design domains. The remainder of this section discusses the conditions that make this approach successful and potentially appropriate for other design problems.

The control design algorithms rely on the encoding in influence graphs of physical field dependencies. Static influence graphs might not be realistic for some physical processes; for example, when material properties vary with temperature. In such cases, it might be necessary to reason with sets of influence graphs; for example, different influence graphs for different temperature bins.

One key piece of physical knowledge leveraged by the control design algorithms is that of locality. Control placement design strives for decoupling, placing controls so that they interfere as little as possible. This allows control parameter optimization
to individually optimize the resulting controls. In addition, control parameter optimization separately considers a control's effects on strongly-influenced nodes and on weakly-influenced nodes. Certain problems, such as heat transfer with highly conductive materials, may not possess strong locality; such problems are less amenable to these approaches.

Control placement design forms groups of control probes with similar effects on the field. This technique relies on continuity of effects: nearby controls have similar effects, unless there are particular constraints due to geometry and material properties. The goal of the control probes is to uncover these constraints. This requires that probes be dense enough, relative to conditions imposed by geometry and material properties, so that groups of probes with similar effects can be uncovered. Otherwise, each probe ends up in its own class, and the decomposition is too dependent on probe placement.

Many physical processes (e.g. heat conduction, gravity, electrostatics, and incompressible fluid flow) obey linear superposition of solutions. The influence graph-based optimization process uses this property to evaluate fields efficiently, based on sums of influences. The influence graph encapsulates other possibly nonlinear irregularities in physical fields, exposing linear dependence on control values.

This chapter considers control design for physical phenomena governed by diffusion processes. Extending this work to address wave phenomena remains as a future research topic.
CHAPTER 5

CONCLUSION

5.1 Thesis Summary

This thesis has described the design and implementation of the Spatial Aggregation Language, a high-level language supporting data interpretation and control tasks for distributed physical systems. The language operators and data types are at a level of abstraction appropriate for programming such applications. They leverage programmer-provided domain-specific knowledge to uncover and exploit multi-level structures in spatially distributed data. The language is supported by a large C++ library and an interpreted, graphical programming environment.

This thesis has also presented a case study application of the Spatial Aggregation Language to decentralized control design. Language operators allow common computational patterns for modeling partial differential equations to be programmed at an appropriate level of abstraction. The influence graph mechanism supports decentralized control design utilizing structural descriptions uncovered in physical data. The influence graph-based control design algorithms decompose a problem domain into minimally-coupled subregions, efficiently evaluate fields, and compare parametric design trade-offs. The algorithms are efficient and yield explainable designs.
5.2 Future Work

The Spatial Aggregation framework was inspired by the common computational structure of a number of interesting applications, and as the language is used in more domains, that computational structure will need to expand. For instance, in interpreting weather data, the language must incorporate operators for finding correspondences between objects; for example, identifying troughs by corresponding sharp turning points in isobars [38]. In interpreting the morphogenesis of diffusion-reaction systems, the language must represent and reason about temporal sequences of objects and aggregates [59].

Other applications might require incorporation of techniques such as uncertain reasoning and learning. Layered abductive reasoning [40] could be incorporated to generate higher-level structures as explanations for lower-level data. Explicitly reasoning about hypothesis interactions and implications. For example, this would allow principled comparisons among a number of possible equivalence classes abstracting a set of data. Similarly, incorporation of learning techniques would allow classifications generated for one set of data to be generalized as classification rules for other sets of data [24].

The actual SAL implementation also has room for additional work. The library establishes a framework of generic interfaces for the language data types. Additional high-level capabilities, such as geometric object intersection and topological equivalence tests, could be layered over the current interfaces. Additional implementations, such as curve and surface patch spatial objects, higher-dimensional neighborhood graphs, and additional spatial indices, could be provided for the interfaces. The
SAL interactive programming environment would benefit from additional CAD-like graphical tools for constructing and manipulating spatial objects.

The Spatial Aggregation Language could benefit from tighter integration with existing programming tools. For example, the spectral partitioning code in Section 4.2 uses an eigenvalue computation on a matrix derived from a neighborhood graph. Seamless integration with numerical libraries is required for such high-level programming. With similar high-level interfaces, SAL could also leverage existing computational libraries containing powerful code for mesh generation, geometric object representation, and spatial indexing. Finally, wrapping up parts of the language as Matlab packages could help make SAL accessible to a broader community.

SAL currently requires an application programmer to recognize appropriate domain knowledge, choose layers of abstraction, and hook up programming elements within each layer of abstraction. Ideally, much of this process could be automated. In the spirit of the Compositional Modeling Language [23], a large library of domain knowledge fragments could be provided, and a programmer could select and mix appropriate pieces. For example, a programmer could choose the heat equation fragment, and a SAL compiler would then automatically be able to choose neighborhood graphs and interaction rules, and exploit knowledge of locality and influence linearity. Another possible approach is to use planning-style reasoning to search through sequences of transformations and their pre- and postconditions, in order to map from an input field to the desired type of high-level structures. For example, in the trajectory bundling application (Section 3.4.1), a planner could recognize that in order to find bundles it must identify neighboring curves with similar limit behavior, and in order to find curves it must find linearly connected points.
In the decentralized control design case study, empirical evidence was provided demonstrating that the structure-based design algorithms perform at least as well as standard approaches, and also support explicit trade-offs between criteria such as communication and control quality. However, no mathematical proofs guaranteeing properties of the control design were presented. It would be interesting to see which properties of the algorithms, if any, are governed by explicit formulas.

The control placement design algorithms seek to decompose a field in order to place controls so that they minimally interfere with each other. Other criteria are also important, and could be combined with this approach. For example, if there is exactly one desired temperature profile, its characteristics could be used to steer control placement to locations where most heat is required.

The control parameter design algorithms deal with temperature regulation by varying control output around some set point. The algorithms naturally extend to transient control, where the goal is to track some desired profile over time. By discretizing controls in time as well as in space, the same properties of locality and linearity hold, and the same trade-offs between communication and control quality can be made. However, additional design criteria (e.g. total error over time vs. maximum error at any point) become important. Furthermore, transient control opens up new avenues of control design; for example, achieving decentralized control with one moving control rather than with a set of stationary controls.

The control design algorithms also explicitly focus on the thermal regulation domain. Since many other systems, including electrostatics, gravity, and incompressible fluid flow, obey the same model, it would be interesting to study application of the techniques developed here to those domains.
The control design algorithms only address one side of the picture: placement and optimization of controls. Equally important is the dual problem of sensor placement. In some cases, such as heat control with readily-available data from an infrared camera, the sensor placement problem need not be addressed. In other cases, however, it has a great impact on the control design. For example, controls might have access only to data from nearby sensors, and sensors might not even be available in some parts of the domain. One simple approach to sensor placement is to co-locate sensors with actuators. Then sensor information could be propagated to controls, perhaps using methods similar to the reduced-communication optimization algorithm in order to trade off between communication frequency and accuracy.

A more sophisticated approach to sensor placement requires reasoning about the information available at various sensor locations; that is, what each potential location reveals about the effects of the controls. A mechanism dual to the influence graph could be defined to encode distributed representations of information available at potential sensor locations. This mechanism could then be utilized to place sensors so as to maximize coverage and minimize overlap, just as with the control placement algorithm.

A related consideration is that of the effect of noise on sensors and the resulting control actions. Standard control techniques could be applied to smooth data over time in order to reduce the effects of noise. However, more powerful techniques could use influence graphs and the information graphs proposed above in order to reason about potential error in the data. For example, outlying data points could be identified by building up a model of the influence hill for a control, and noticing when a data point does not conform to the appropriate shape. The influence graph
mechanism could also be extended, in a manner similar to the reduced communication optimization algorithm, to reason about the effects of sensor uncertainty. That is, there is a curve trading off error in sensor state and error in control analogous to the curve trading off frequency of sensor state update and error in control.

5.3 The Big Picture

We are entering an age where many exciting applications, ranging from smart buildings to self-diagnosing photocopy machines to airplanes steered by micro-flaps, require understanding and controlling massive, spatially-distributed physical systems. There are many challenges in programming such data interpretation and control applications, and existing tools are not sufficient. The Spatial Aggregation Language represents a step towards the development of powerful programming environments for these tasks: it provides high-level data types and operators in a decentralized framework, it uses explicit representations of physical knowledge, and it bridges local and global representations through multiple layers of abstraction. The control design algorithms presented here also exemplify many of the characteristics desirable for such applications: by decomposing and decentralizing they are massively scalable, by reasoning in terms of spatial structures they provide explainable design decisions, and by utilizing physical knowledge they expose trade-offs among desirable design properties.
APPENDIX A

CURRENT SAL C++ LIBRARY COMPONENTS

The current implementation of the SAL library contains about 1.5 megabytes of C++ source code. The following tables list the current contents of the library, in terms of interface and implementation classes. Indented below each generic interface class are more specific subclass interfaces and implementations. The leaves are implementations or interfaces with only default implementations.

In addition to classes supporting the data types and operators defined in this thesis, there are also a number of Inspector classes for reading, writing, and plotting objects. Inspectors make use of the graphical widgets provided by Tcl/Tk [60] and the 3-d plotting package Geomview [77].
### Table A.1: Spatial object classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SObject</td>
<td>Structure of a physical object.</td>
</tr>
<tr>
<td>CComplex</td>
<td>Cell complex</td>
</tr>
<tr>
<td>Ngon</td>
<td>(N)-gon with interior (2-d)</td>
</tr>
<tr>
<td>Ngraph</td>
<td>Segments corresponding to adjacencies between points in an ngraph.</td>
</tr>
<tr>
<td>Mesh</td>
<td>Ngons bounded by segments of ngraph cell complex.</td>
</tr>
<tr>
<td>Delaunay</td>
<td>Triangular elements based on Delaunay ngraph.</td>
</tr>
<tr>
<td>RNG</td>
<td>Polygonal elements based on relative neighborhood graph.</td>
</tr>
<tr>
<td>MST</td>
<td>Segments based on minimal spanning tree ngraph.</td>
</tr>
<tr>
<td>Nhedron</td>
<td>(N)-hedron with interior (3-d)</td>
</tr>
<tr>
<td>Nline</td>
<td>Sequence of contiguous segments (1-d)</td>
</tr>
<tr>
<td>Point</td>
<td>Point (0-d)</td>
</tr>
<tr>
<td>Segment</td>
<td>Closed line segment (1-d)</td>
</tr>
</tbody>
</table>

### Table A.2: Geometric object classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geom</td>
<td>Maintains properties of an object in a metric space.</td>
</tr>
<tr>
<td>CComplex.Geom</td>
<td>A geom whose structure is a cell complex.</td>
</tr>
<tr>
<td>Ngon</td>
<td>Ngon geometric properties.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Ngon geometric properties based on vertex geometries for rectangularly-shaped geometry.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Ngon geometric properties based on vertex geometries for triangle-shaped geometry.</td>
</tr>
<tr>
<td>Ngraph</td>
<td>Ngraph geometric properties based on vertex geometries.</td>
</tr>
<tr>
<td>Nline</td>
<td>Nline geometric properties based on vertex geometries.</td>
</tr>
<tr>
<td>Nhedron</td>
<td>Nhedron geometric properties.</td>
</tr>
<tr>
<td>Point</td>
<td>A point with coordinates.</td>
</tr>
<tr>
<td>1D</td>
<td>Embedded in a one-dimensional coordinate space.</td>
</tr>
<tr>
<td>2D</td>
<td>Embedded in a two-dimensional coordinate space.</td>
</tr>
<tr>
<td>ND</td>
<td>Embedded in an (n)-dimensional coordinate space.</td>
</tr>
<tr>
<td>Segment</td>
<td>Segment geometric properties based on vertex geometries.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Space</td>
<td>Collection of objects</td>
</tr>
<tr>
<td>Adjacency</td>
<td>Directed link between objects in an ngraph.</td>
</tr>
<tr>
<td>Metric_Space Function</td>
<td>Space with metric function</td>
</tr>
<tr>
<td>Induced</td>
<td>User-specified metric</td>
</tr>
<tr>
<td>Vertices</td>
<td>Maps objects to user-specified representatives in another metric space.</td>
</tr>
<tr>
<td>Coord_Space</td>
<td>Metric space based on Coord_System</td>
</tr>
<tr>
<td>Centroid</td>
<td>Measures distance between object centroids.</td>
</tr>
<tr>
<td>2D_Array</td>
<td>Indexes 2-d points in an array for (O(r^2)) nearness search with radius (r).</td>
</tr>
<tr>
<td>2D_Buckets</td>
<td>Indexes 2-d points in an array of buckets for (O(r^2 + m)) nearness search with radius (r) that returns (m) objects.</td>
</tr>
<tr>
<td>3D_Array</td>
<td>Indexes 3-d points in an array for (O(r^3)) nearness search with radius (r).</td>
</tr>
<tr>
<td>KD_Tree</td>
<td>Indexes (k)-d points in a (k)-d tree for (O(n^{1-1/k} + m)) nearness search that returns (m) of the (n) points.</td>
</tr>
<tr>
<td>Sequence_CircList</td>
<td>Circular list of objects.</td>
</tr>
<tr>
<td>Set_FastSearchList</td>
<td>Set of objects with (O(\log n)) membership test.</td>
</tr>
<tr>
<td>Set_Linked_Eq</td>
<td>Set of objects with (O(n)) membership test based on pointer equality.</td>
</tr>
<tr>
<td>Vector_Growing</td>
<td>Dynamically sized vector of objects with integer indexes.</td>
</tr>
<tr>
<td>Coord_System</td>
<td>Coordinate system measuring distances between coords.</td>
</tr>
<tr>
<td>Euclidean</td>
<td>(d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \ldots})</td>
</tr>
<tr>
<td>Manhattan</td>
<td>(d(p, q) =</td>
</tr>
<tr>
<td>Coords</td>
<td>Tuple of numbers.</td>
</tr>
<tr>
<td>1D</td>
<td>One number.</td>
</tr>
<tr>
<td>2D</td>
<td>Two numbers.</td>
</tr>
<tr>
<td>ND</td>
<td>(N) numbers.</td>
</tr>
</tbody>
</table>

Table A.3: Space classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Function Map</td>
<td>Maps from domain metric space to feature metric space. User-specified functions access features. Maintains explicit map.</td>
</tr>
</tbody>
</table>

Table A.4: Field classes.

184
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ngraph</td>
<td>Forms adjacencies among objects.</td>
</tr>
<tr>
<td></td>
<td>Neighbor is the best neighbor in a base ngraph, according to a provided metric.</td>
</tr>
<tr>
<td>Best</td>
<td>Neighbors are the n-th closure of a base ngraph.</td>
</tr>
<tr>
<td>Closure</td>
<td>Neighbors are the union, intersection, or difference of base ngraphs.</td>
</tr>
<tr>
<td>Combination</td>
<td>Neighbors are those adjacencies in a base ngraph satisfying a predicate.</td>
</tr>
<tr>
<td>Explicit</td>
<td>Explicit functions let the user build the adjacencies.</td>
</tr>
<tr>
<td>Filter</td>
<td>Neighbors are those adjacencies in a base ngraph satisfying a predicate.</td>
</tr>
<tr>
<td>Grid</td>
<td>Builds adjacencies for rectilinearly-positioned points.</td>
</tr>
<tr>
<td>Inverse</td>
<td>Adjacencies are in the opposite direction from a base ngraph.</td>
</tr>
<tr>
<td>K_nearest</td>
<td>Neighbors are the k nearest objects; cost depends on spatial index.</td>
</tr>
<tr>
<td>Mesh_Dual</td>
<td>Nodes are elements in a mesh complex; neighbors are adjacent elements in the complex.</td>
</tr>
<tr>
<td>Near</td>
<td>Neighbors are everything within a radius, either fixed or object-dependent; cost depends on spatial index.</td>
</tr>
<tr>
<td>Reachable</td>
<td>Neighbors are those reachable from a given set of nodes within a given number of adjacencies.</td>
</tr>
<tr>
<td>Subgraph</td>
<td>The subset of the graph defined on a given space.</td>
</tr>
<tr>
<td>Substructure</td>
<td>Neighbors based on ngraph for substructure of objects.</td>
</tr>
<tr>
<td>Symmetric</td>
<td>Adjacencies added or removed to turn a directed ngraph into an undirected one.</td>
</tr>
<tr>
<td>CComplex</td>
<td>Builds ngraph cell complex corresponding to adjacency structure.</td>
</tr>
<tr>
<td>MST</td>
<td>Minimal spanning tree built in time $O(a \log a)$ for base ngraph with a adjacencies.</td>
</tr>
<tr>
<td>Mesh</td>
<td>Builds mesh cell complex corresponding to adjacency structure.</td>
</tr>
<tr>
<td>Delaunay</td>
<td>$2-d O(n \log n)$ point triangulation.</td>
</tr>
<tr>
<td>RNG</td>
<td>$2-d O(n \log n)$ relative neighborhood graph.</td>
</tr>
</tbody>
</table>

Table A.5: Ngraph classes.
Class Description

**Classifier**
- **Explicit**
- **First_N**
- **Transitive**
  - Partitions a space into equivalence classes.
  - Explicit functions let the user build the classes.
  - Follow a predicate through a neighborhood graph, collecting at most \( n \) objects into each group; \( O(a) \) for \( a \) adjacencies.
  - Follow a predicate transitively through a neighborhood graph; \( O(a) \) for \( a \) adjacencies.

**Space_Predicate**
- **And**
- **Bin**
- **Feature_Distance**
- **Not**
- **Or**
- **Xor**
  - Given a space, returns whether or not it satisfies a predicate.
  - Takes the conjunction of two predicates.
  - Tests if the field features of all objects fall in the same bin.
  - Tests if the maximum distance between field features of all objects satisfies a threshold.
  - Negates a predicate.
  - Takes the disjunction of two predicates.
  - Takes the exclusive-or of two predicates.

**Adjacency_Predicate**
  - Specialization of space predicate where the space is an adjacency.
  - Efficient implementations of space predicates.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstractor</strong></td>
<td>Maintains mapping between equivalence classes and higher-level objects.</td>
</tr>
<tr>
<td><strong>Disaggregate</strong></td>
<td>Applies a top-down disaggregation operator to build the mapping.</td>
</tr>
<tr>
<td><strong>Explicit</strong></td>
<td>Explicit functions let the user build the mapping.</td>
</tr>
<tr>
<td><strong>Redescribe</strong></td>
<td>Applies a bottom-up redescribe operator to build the mapping.</td>
</tr>
</tbody>
</table>

**Redescribe_Op**
- **Any**
- **BBox**
- **Path_Nline**
- **Points_Hull**
- **Points_Segment**
- **Segments_Ngon**
- **Segments_Nline**
  - Classes to objects function for Abstractor_Redescribe.
  - Chooses a member of the class.
  - Builds a bounding box for the centroids of a class of 2-D geometric objects.
  - Builds an nline for a class of points based on connectivity in an ngraph.
  - Builds a convex hull for a class of 2-D points.
  - Builds a segment connecting a class of two points.
  - Builds an ngon connecting a class of segments.
  - Builds an nline connecting a class of segments.

Table A.6: Classifier classes.

Table A.7: Abstractor classes.
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspector</td>
<td>Allows inspection of SAL objects (geometric objects, neighborhood graphs, etc.).</td>
</tr>
<tr>
<td>Inspector_Dialog</td>
<td>Textual input and output.</td>
</tr>
<tr>
<td>Tk</td>
<td>Tcl/Tk dialog box.</td>
</tr>
<tr>
<td>Inspector_Plot</td>
<td>Graphical output and modification.</td>
</tr>
<tr>
<td>Geomview</td>
<td>3-d plots communicated to separate Geomview process; graphical selection of objects.</td>
</tr>
<tr>
<td>PS</td>
<td>Postscript graphics saved in a file.</td>
</tr>
<tr>
<td>Tk_Canvas</td>
<td>Plot to Tcl/Tk canvas widget; graphical selection of objects.</td>
</tr>
<tr>
<td>Tk_Pixels</td>
<td>Raw pixel plot to Tcl/Tk window; coordinates for mouse clicks.</td>
</tr>
<tr>
<td>Inspector_Read</td>
<td>Textual input.</td>
</tr>
<tr>
<td>In_File</td>
<td>From input file.</td>
</tr>
<tr>
<td>In_String</td>
<td>From C character string.</td>
</tr>
<tr>
<td>Inspector_Write</td>
<td>Textual output.</td>
</tr>
<tr>
<td>Out_File</td>
<td>To output file.</td>
</tr>
<tr>
<td>Out_String</td>
<td>To C character string.</td>
</tr>
<tr>
<td>Out_Tk_Text</td>
<td>To Tcl/Tk text widget.</td>
</tr>
</tbody>
</table>

Table A.8: Inspector classes.
APPENDIX B

SAL IMPLEMENTATION DETAILS

This appendix serves as a user manual for the current implementation of the SAL library and interpreter. It describes details in the implementation and use of the library and interpreter relevant to users actually programming with one or the other. This discussion pertains primarily to the version of the library and interpreter for various flavors of the Unix(tm) operating system. Iván Ordóñez has ported parts of the implementation to MacOS(tm).

B.1 Library Implementation Conventions

A number of design decisions made in implementing the library affect the style of programming for SAL library-based programs.

B.1.1 Class Hierarchy

As mentioned in Section 3.3, the library obeys a strict interface/implementatio distinction. Interface classes specify required functionality; multiple implementation subclasses of an interface can provide that functionality in different ways. This is implemented in C++ by specifying abstract base classes (e.g. Ngraph) with pure virtual functions (e.g. neighbors) and then deriving subclasses (e.g. Mesh_Delaunay) that provide code for those virtual functions.
Only essentially implementation-specific functions are specified in the interface classes. Additional implementation-independent functions are provided as global capabilities that can act on an instance of any implementation meeting a particular interface. For example, the function to return an object’s neighbors in a neighborhood graph depends on the neighborhood relation defined by an implementation. However, a function to plot a neighborhood graph can be applied to any implementation by layering it over the implementation-specific neighbor query.

The *Object* class is the root class for the entire library. This class specifies a *new_object* function that each implementation class must define to create a new instance of the given type. This supports the use of prototype instances to create new instances, as described below.

C++ templates are used only for generic container classes (e.g. sets, and sequences); they are generally instantiated with pointer types (e.g. TSet<Point *>). The rest of the library relies on type-casting and run-time type checking. That is, rather than specifying statically that a neighborhood graph aggregates only points by instantiating a template Ngraph<Point *>, the library relies on the programmer to use the instance consistently, treating an aggregated object as a point only when appropriate. This is a trade-off between type safety for the more generic classes, and simplicity for the more complex SAL-specific classes. In most cases, knowing the exact type of the objects is not important; for example, it is often sufficient simply to aggregate a set of spatial objects in a neighborhood graph and use generic interface-level operations on the objects.
B.1.2 Functions

A function is not required to ensure that it is called legally — that is the duty of the caller. If a function is invoked illegally (e.g. ask for the neighbors of an object not in a neighborhood graph), the function's action is unspecified. Checking implementations can wrap existing implementations with tests for safety. Such implementations could be used during program design to help debug and then easily be removed for faster execution of the final program [79].

Some functions take example prototype instances, to serve as the basis for creating more instances. For example, an ngraph can take a prototype for the space into which neighbors should be stored. Then one program might supply a simple set prototype, while another might supply an indexing space prototype supporting efficient spatial queries on an object's neighbors. Typically, defaults are provided using the simplest implementations; the user only provides a prototype in order to tailor the run-time performance of a program.

The SAL library specifies Op classes as a uniform interface for passing functions to other functions. Different op interface classes specify different numbers of arguments and return types; template parameters flesh out the signatures of the functions. For example, \( \text{Op}_2< A_1, A_2, R > \) is a template class for functions that take two arguments (types \( A_1 \) and \( A_2 \)) and return one value (type \( R \)), while \( \text{Op}_2\_\text{void}< A_1, A_2 > \) also takes two arguments but does not return a value. Each derived op implementation class defines a function of the corresponding signature. Table B.1 summarizes the op interface classes.
- **Op\textsubscript{n}:** \(n + 1\) template parameters; call function taking \(n\) arguments of the first \(n\) parameter types and returning a value of the final parameter type. For example, a subclass of Op\textsubscript{1}\textlangle Ngraph *, Space *\textrangle could wrap up the Ngraph neighbors function.

- **Op\textsubscript{n}.void:** \(n\) template parameters; call function taking \(n\) arguments of the parameter types and returning no value. For example, a subclass of Op\textsubscript{2}.void\textlangle Ngraph *, Space *\textrangle could wrap up the Ngraph aggregate function.

Table B.1: Op classes.

### B.1.3 Memory

The SAL library incorporates Hans-J. Boehm's garbage collection package \cite{8} to handle memory deallocation. The compiler flag "SALGC," defined true in the standard SAL makefiles, specifies that all classes inheriting from the base \texttt{Object} class should be garbage collected. Note that explicit object deletion can still be performed in application programs, if one derives pleasure from tracking down memory bugs.

When an object creates another object as the response to a query, the creator is assumed to "own" the created object unless otherwise indicated. In particular, when the creator is destroyed, the created object can also be destroyed. For example, an ngraph owns the sets of neighbors it returns; when the ngraph is destroyed, those sets are no longer valid.

### B.1.4 Iteration

While the language is specified in terms of collection-oriented functions such as \texttt{map} and \texttt{select}, the library (which runs on a single sequential processor) implements these functions in terms of \texttt{Iterators} on collections. An iterator is a class with the
ability to step through a collection one element at a time. Thus, for example, *map* can be programmed by iterating through a space one element at a time, applying a function to each element, and storing the result in the output space. Subclasses support additional functionality, such as reversing direction and starting from a given position in a sequence.

### B.2 Using the Library

The SAL distribution provides and uses several tools to support the development of application programs.

#### B.2.1 Makefile

Application programs must correctly include library files and link with various libraries, including SAL and the garbage collector. To simplify this process, the distribution includes a standard makefile with the example programs. This makefile uses the environment variable “SALLIB” to find the root of the SAL library in order to include the proper files and link to the proper libraries.

#### B.2.2 Ops

In order to enhance the readability of application programs that use Op classes, a Perl script preprocesses specially-formatted functions into classes whose instances can be passed around as ops. The script is invoked with the name of the input file and the output C++ file; the standard SAL makefile transforms a “.oc” file to a corresponding “.c++” file.

The input code flags functions to be preprocessed with a “#Op” tag. Optional parameters to the function-creating routine follow the “#Op” tag, enclosed in square
Table B.2: Syntax for preprocessor-based operation passing.

brackets and declared like a standard C++ argument list. Actual values for these parameters are passed when a new instance of the op is created; they are then visible within the scope of the function body. Table B.2 shows the syntax for the preprocessor input.

The output contains a class "Op.name" for each tagged function "name." Constructor arguments and state variables for the class are derived from the parameter list. If there are no parameters, the script creates an instance of the class, with the given name. Table B.3 shows preprocessor output corresponding to Table B.2, assuming that the return types are not void.

Table B.4 shows a simple example of using the preprocessor to wrap up two functions (without and with parameters) to pass to a map function.
// No parameters
class Op_name : public Op_na<arg-type-1, arg-type-2, ..., arg-type-na, return-type>
{
    public:
    return-type call(arg-type-1 arg-1, arg-type-2 arg-2, ..., arg-type-na arg-na)
    {
        body
    }
};
Op_name *name = new Op_name;

// Parameters
class Op_name : public Op_na<arg-type-1, arg-type-2, ..., arg-type-na, return-type>
{
    public:
    param-type-1 param-1;
    param-type-2 param-2;
    ...
    param-type-np param-np;

    Op_name(param-type-1 init-1, param-type-2 init-2, ..., param-type-np init-np)
    {
        param-1 = init-1;
        param-2 = init-2;
        ...
        param-n = init-n;
    }
    return-type call(arg-type-1 arg-1, arg-type-2 arg-2, ..., arg-type-na arg-na)
    {
        body
    }
};
// No parameters
#0p
int add_one_to(int i)
{
    return i+1;
}

// One parameter: how much to add to the argument
#0p [int j]
int add_j_to(int i)
{
    return i+x;
}

// A container created elsewhere
Container<int> *ints;

// Add 1 to each element of ints
ints2 = map(ints, add_one_to);

// Create an op that will add 2 to its argument
Op_1<int, int> *add_two_to = new Op_add_j_to(2);

// Add 2 to each element of ints
ints3 = map(ints, add_two_to);

Table B.4: Example of preprocessor-based operation passing.
B.3 Extending the Library

The files comprising the library are divided in a directory structure mirroring the main hierarchy of classes (e.g. separate directories for spatial objects, neighborhood graphs, and fields). The \textit{caps} directory stores high-level capabilities, while the \textit{include} directory collects pointers to all the include files. Additional implementation-specific support files are placed in the support directories. A root makefile descends through the hierarchy, invoking directory-specific makefiles.

In order to extend the library, files should be placed in the appropriate directory and the corresponding makefile updated. The include directory should have a link to the header.

B.4 Using the Interpreter

B.4.1 Interaction

The SAL interpreter uses the GNU readline package as the basis for getting user commands. This package automatically handles history maintenance. For example, “!!” repeats the previous command, “\texttt{Ctrl-p}” goes back to the previous command (further back upon repetition), and “\texttt{Ctrl-r}” searches back through the interaction history for a command containing a specified string. It also defines a number of Emacs-style key bindings for manipulating the current input line. For example, “\texttt{Ctrl-a}” goes to the front of the input line, “\texttt{Ctrl-f}” moves forward a character, and “\texttt{Ctrl-k}” kills the input to the right of the current cursor position.

By default, SAL catches segmentation faults and interrupts (“\texttt{Ctrl-c}”), and returns to the top-level input loop. Running the interpreter with the flag “debug” instructs SAL not to catch these signals.
B.4.2 Functions

Most of the functionality of the interpreter is simply calls to corresponding SAL library functions, such as aggregate, classify, and redescribe. In addition to these SAL-specific function calls, the interpreter provides a few generic primitive functions for interacting with the interpreter (e.g. to load a program or to examine a function’s documentation). Table B.5 details these commands.

B.5 Extending the Interpreter

The lib subdirectory of the interpreter contains code implementing the functions supported by the library. The functions consist mainly of wrapper calls to the SAL library, packaging and unpack packaging argument lists and return values. A set of scripts helps automate the process of extending the interpreter’s functionality.

B.5.1 Defining a New Type

The interpreter type-checks arguments to function calls. In order to do this, the interpreter needs to know the structure of the library class hierarchy. Adding a new type requires placing it correctly in the hierarchy and generating a meta-type for use by the type checker. This process is simplified by the gen_type script, which inputs a file describing the types. The script separately handles two kinds of types, one for a class hierarchy, and one for function types. Table B.6 details the syntax of the input file lines for these two types.

The script outputs a file to register the class and function meta-types; the file should be linked to the interpreter and its registration function invoked in the interpreter’s initialization routine.
- **abort** : () → ()
  Returns to the interpreter top-level.

- **apropos** : (String n1, String n2, ...) → ()
  Lists functions/variables whose documentation contains any of the names.

- **cd** : (String d) → ()
  Changes the working directory to d.

- **exists** : (Object o) → (Boolean b)
  Is o a real object (as opposed to a null pointer)?

- **help** : (String n) → ()
  Prints the documentation for the named function/variable: if none is given, provides general information.

- **source** : (String f) → ()
  Executes the commands in the file.

- **source_halt** : () → ()
  Halts the current sourcing.

- **source_step** : () → ()
  Begins stepping the current source.

- **step** : (String f) → ()
  Allows the user to bring the commands in the file to the prompt with “Ctrl-g.”

- **step_halt** : () → ()
  Halts the current stepping.

- **step_run** : () → ()
  Reads in the remaining input from the currently stepped file.

- **quit** : () → ()
  Exits the interpreter.

Table B.5: Primitive interpreter operations.
(a) Class hierarchy
class-name : superclass-name

(b) Function types
type-name : function (arg-type-1 arg-1, ..., arg-type-n arg-n) -> (return-type return)

Table B.6: Syntax of input file specifying types to be used in the SAL interpreter.

function-name = extern function (arg-type-1 arg-1, ..., arg-type-n arg-n) -> (return-type return) : documentation

Table B.7: Syntax of input file specifying functions to be used in the SAL interpreter.

An additional step must be taken to create Ops to wrap up functions passed as arguments. The gen.ops script takes the same input file as gen.type and generates the appropriate Ops. This file will be included as a header for the function wrappers described below.

B.5.2 Defining a New Function

Functions to be supported by the interpreter must be explicitly registered, with a proper type signature specified and a pointer to the function code provided. Again, scripts provided with the interpreter help automate the process. The input file describes each function with a lines of the form shown in Table B.7.

The gen.register script uses such an input file to generate a header file and code file that inform the interpreter of the function’s name, type signature, documentation
(used by the help and apropos commands), and code pointer. The code file should be linked to the interpreter and its registration function invoked in the interpreter’s initialization routine.

The gen_wrappers script uses the same input file to generate wrappers converting from an interpreter-based argument vector to C++ objects to be passed to a library function. The conversions involve boxing/unboxing numbers (the interpreter stores everything as an Object, while some library functions use, for example double-precision numbers), and wrapping/unwrapping function pointers as Ops. While code for the conversions is done automatically, the actual body of the wrapper is left blank and must be filled in with a call to the appropriate SAL library function.
BIBLIOGRAPHY


201


202


