Spatial Aggregation: Language and Applications[*]

Christopher Bailey-Kellogg and **Feng Zhao**
Computer and Information Science
The Ohio State University
2015 Neil Ave.
Columbus, OH 43210 U.S.A.
{kellogg,fz}@cis.ohio-state.edu

Kenneth Yip
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139 U.S.A.
yip@martigny.ai.mit.edu

Abstract

This paper describes the spatial aggregation language and its applications. Spatial aggregation comprises a framework and a mechanism for organizing computations around image-like, analogue representations of physical processes in data interpretation and control tasks; it transforms a numerical input field to successively higher-level descriptions by applying a small, identical set of operators to each layer given a metric, neighborhood relation and equivalence relation.

The spatial aggregation language provides two abstract data types (ADTs) — neighborhood graph and field — and a set of interface operators for constructing the transformations of the field. The language consists of a library of component implementations from which a user can mix-and-match and specialize for a particular application. The modular design of the ADTs supports language extensions and user control over tradeoffs such as efficiency vs. generality.

We illustrate the use of the language with examples ranging from region growing in image analysis to trajectory grouping in dynamics interpretation. Programs for these different task domains can be written in a modular, concise fashion in the spatial aggregation language. The language allows users to isolate and express important computational ideas while hiding low-level details.

Content Areas: Qualitative Reasoning, geometric/spatial reasoning, programming language, ontologies, applications.

Introduction

Effective reasoning about a physical system requires an appropriate mapping from the system characteristics to abstractions that match the requirements of the task at hand. Spatial aggregation organizes computations around image-like, analogue representations of physical processes in data interpretation and control tasks (Yip & Zhao 1996). In Qualitative Physics, three ontological abstractions are widely used: device, process, and constraint. Spatial aggregation introduces a new ontological abstraction, the *field ontology*, to unify many reasoning tasks involving the image-like analogue representations such as the velocity field for fluid motion, phase space for dynamical systems, and configuration space for mechanism analysis.

The input to spatial aggregation is a data massive, numerical field. [1] The desired output is a high-level, parsimonious description of the structure and behavior of the physical process that the field represents. To bridge the semantic gap between the analogue input field and the final symbolic description, spatial aggregation introduces layers of intermediate structures called spatial aggregates to capture spatial adjacencies among objects of the field at multiple spatial and temporal scales. A spatial aggregate is constructed from a metric, a neighborhood relation and an equivalence relation supplied by a user according to the objective of computation. Spatial aggregation transforms the input field to successively higher-level descriptions by applying a small, identical set of operators to each layer of the spatial aggregates.

The spatial aggregation framework grows out of a class of problem solvers, KAM (Yip 1991), MAPS (Zhao 1994) and HIPAIR (Joskowicz & Sacks 1991), that derive their power primarily from perceptual operators on analogue representations, and only secondarily from search and analytical methods. These programs have exhibited expert performance on difficult problems in hydrodynamics, nonlinear control, and engineering mechanism analysis. Spatial aggregation abstracts the common computational structure

[*] FZ is supported by an NSF National Young Investigator Award CCR-9457802, an Alfred P. Sloan Foundation Research Fellowship, a grant from Xerox Palo Alto Research Center, and an NSF grant CCR-9308639. CBK is supported by FZ's NSF NYI grant CCR-9457802. KY is supported by an NSF National Young Investigator Award ECS-935777.

[1] A field maps one continuum to another. Examples include velocity field ($R^3 \to R^3$), temperature field ($R^3 \to R^1$), image field ($R^2 \to R^1$), and vector field ($R^n \to R^n$).

and a set of generic operators from these problem solvers. It can also apply to a wide variety of other task domains such as image analysis and geographic information databases applications. The generic operators of spatial aggregation can be viewed as a particular instantiation of Ullman's "visual routines" for visual information processing tasks (Ullman 1984; Mahoney 1995).

Other researchers have developed related frameworks and systems for reasoning about spatial, analogue representations of physical world. For example, Forbus et al. developed the Metric Diagram/Place Vocabulary (MD/PV) framework for qualitative spatial reasoning (Forbus, Nielsen, & Faltings 1991). Chandrasekaran and Narayanan proposed a direct analogue simulation of elementary mechanics problem using a diagrammatic representation (Chandrasekaran & Narayanan 1990). In comparison, the spatial aggregation framework comprises multi-layer spatial aggregates with identical computational structure at each layer and focuses on the problem of recovering structures from numerical fields.

This paper describes the spatial aggregation language and its implementation, and the development of applications in the style of spatial aggregation. The spatial aggregation language allows users to isolate the important computational ideas in different problem domains and provides primitives and means of abstraction to express these ideas concisely while hiding low-level details. More specifically, the language provides two abstract data types (ADTs) — neighborhood graph and field — and a set of interface operators for constructing the transformations of the field. The language includes a library of component implementations from which a user can mix-and-match and specialize for a particular application. The modular design of the ADTs supports language extensions and user control over tradeoffs such as efficiency vs. generality. The language can be extended by providing additional operators or new implementations for the ADTs. We illustrate the use of the language with examples ranging from region growing in image analysis to trajectory grouping in dynamics interpretation. We show that programs for these different task domains can be written in a modular, concise fashion in the spatial aggregation language.

Overview

Given an input field, spatial aggregation constructs a neighborhood graph (N-graph) from primitive objects of the field, explicates their spatial adjacencies, and forms equivalence classes of these objects using an equivalence relation determined by the objective

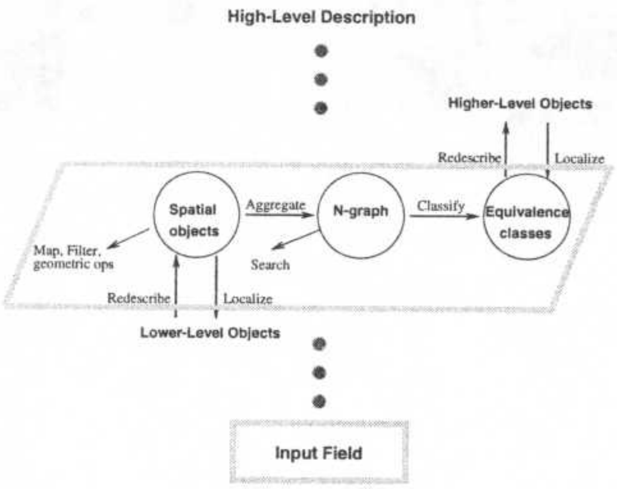

Figure 1: Spatial aggregation: the lower-level objects from the numerical input field are transformed into higher-level objects through a sequence of operations available in the language. The higher-level objects then become the input to another level of spatial aggregation where the identical set of operators apply.

of the task. An equivalence class can be redescribed as a primitive object at a higher level if necessary, and the identical steps of aggregation to form a new N-graph and classification to form equivalence classes apply with a new metric, neighborhood relation, and equivalence relation. This iteration terminates when the desired behavioral and structural description can be readily derived from the N-graph. The N-graph and field serve as computational glue for the operations that search, transform, and filter the spatial objects. Figure 1 illustrates the data flow in the main operations of the language at each level of spatial aggregation.

Spatial aggregation represents primitive objects of a physical process or system with *spatial objects*. For instance, a spatial object might describe a state of a dynamical system — a point and its direction of movement in an n-dimensional phase space spanned by the state variables. A spatial object comprises a geometric description and a feature description. The geometric description is specified in a *metric space* defining distances between geometric primitives. The feature description belongs to one or more *feature spaces*. For example, in image analysis, a pixel spatial object uses the pixel location as the geometric description and the associated brightness value as the feature description. Likewise, a region spatial object defines a geometric region in an image and an average

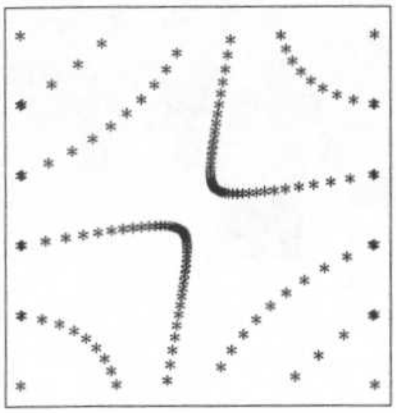

Figure 2: Trajectory bundling operation in phase space dynamics interpretation: the input field consists of states as points in phase space.

or minimum/maximum brightness value of the constituent pixels. In a meteorology application, each spatial object specifies a location in space and a temperature, barometric pressure, and air flow velocity in feature space. The distance between values in a feature space represents how different the corresponding spatial objects are. Spatial aggregation forms neighborhood graphs for spatial objects using the geometric description and groups the spatial objects using similarity or proximity measures in feature space. For example, spatial aggregation could group text with the same font, using a feature space defined by font characteristics. In a mechanism analysis system, it could group configurations in a configuration space.

As an example of how spatial aggregation provides organizational principles and building blocks to facilitate the development of programs for engineering problems, consider an interpretation task in dynamical system analysis. The input is a field of sampled states as points in phase space shown in Figure 2. The objective is to group the states into trajectories and then trajectories into trajectory bundles that share similar limit behaviors, as shown in Figure 6 and Figure 9 respectively.

The first step, *aggregation*, forms a neighborhood graph using a neighborhood relation to explicitly indicate pairs of adjacent spatial objects. Different applications require different neighborhood relations. In the trajectory interpretation application, a minimal spanning tree (MST) is appropriate; other applications use Delaunay triangulations, nearness criteria, and so forth. The spatial aggregation code shown in Figure 3 uses the operator `aggregate` to compute the neighborhood graph (the argument `points-ngraph-fac` —

```
(define points-ngraph
  (aggregate input-field points-ngraph-fac))
```

Figure 3: Aggregation of trajectory points

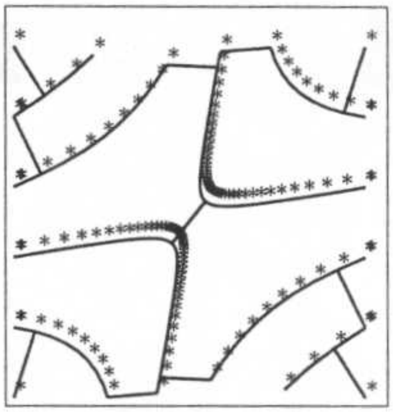

Figure 4: A neighborhood graph (MST) for the points.

constructed from language library components — specifies how to build an MST). Figure 4 shows the result. The operator `aggregate` allows the user to focus on choosing a good neighborhood relation while hiding implementation details.

The next main step, *classification*, forms equivalence classes of neighboring spatial objects according to their similarity in the feature space. In the trajectory interpretation example, a point can be considered similar to a neighbor if their separation is not significantly longer than the distances separating other nearby neighbors. In the code of Figure 5, `classify` forms equivalence classes of points from the MST, `points-ngraph`, by deleting edges that are too long according to the threshold `*point-distance-tol*`; the result is shown in Figure 6. Other classification mechanisms, discussed later in the paper, use consistency predicates to test the classes. The operator `classify` allows the user to select an appropriate equivalence relation and a classification mechanism.

The third main step of spatial aggregation, *redescribing*, maps equivalence classes of objects at one level to single higher-level objects at the next level. In the trajectory interpretation example, each equivalence class of points becomes a single trajectory object; Figure 7 shows the spatial aggregation code. The `redescribe` operator shifts the level of abstraction so that the aggregation process can repeat at a higher level. The inverse of redescribing is *localizing*, which maps each higher-level object to the equivalence class of constituent objects at the lower level.

```
(define point-classes
  (classify
   points-ngraph points *point-distance-tol*))
```

Figure 5: Classification of trajectory points

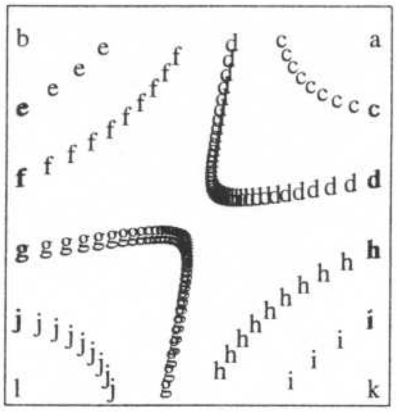

Figure 6: Points grouped into trajectories labeled by a through l respectively.

To group trajectories into trajectory bundles, the same process repeats, using the operators `aggregate`, `classify`, and `redescribe`. The only differences are in the metric, neighborhood relation, and equivalence relation: trajectories are aggregated into a neighborhood graph where the neighborhood is defined by a sphere of some fixed radius, and neighboring trajectories are bundled using an equivalence relation comparing corresponding vectors along trajectories. Figure 8 shows the spatial aggregation code, and Figure 9 shows the result. The aggregation process can repeat at a even higher level if necessary. [2]

As the example demonstrates, programs written in the spatial aggregation language are modular, using a common data structure (neighborhood graph) and an identical set of generic operators (see Figure 10). They are concise and make explicit the important computational characteristics of the problem: neighborhood and equivalence relations.

Additional operators are available for manipulating

[2]In control applications, the trajectory bundles are further aggregated to form reachability graphs; see (Zhao 1994).

```
(define trajs
  (redescribe point-classes traj/create))
```

Figure 7: Redescription of point classes as trajectories

```
;;; Aggregate the trajectories.
(define traj-ngraph
  (aggregate trajs traj-ngraph-fac))

;;; Form equivalence classes.
(define traj-bundles
  (classify
   traj-ngraph trajs *vector-similarity-tol*))
```

Figure 8: Aggregation and classification of trajectories

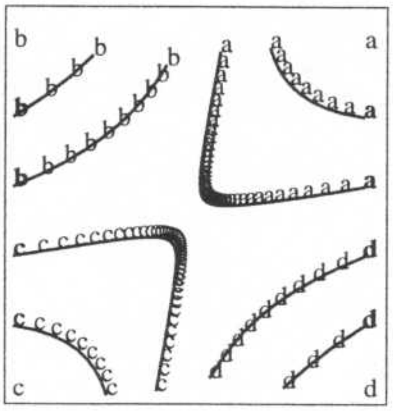

Figure 9: Trajectories grouped into bundles labeled a through d respectively.

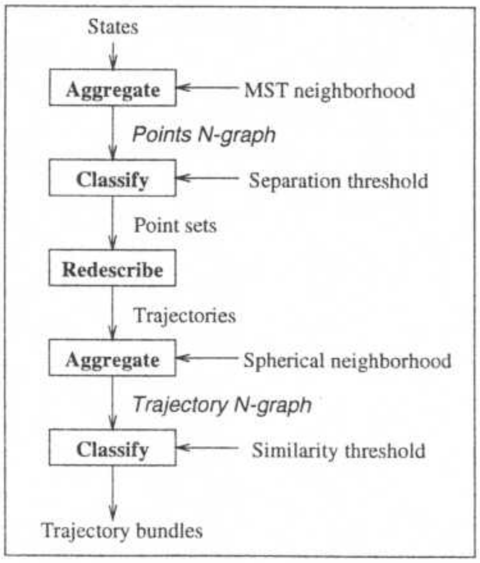

Figure 10: The flow chart of the phase space interpretation example.

Spatial Aggregation Language Features

- Data types:

  `Ngraph` and its constructors, accessors, modifiers. Examples of `ngraph`: 4-adjacency, MST, and Voronoi diagram.

  `Field` and its constructors, accessors, modifiers. Examples of `field`: array, grid, and k-d tree.

- Interface operators:

  `aggregate, classify, redescribe, localize, search`

  A user must specify the neighborhood relation, field metric, and equivalence relation for these operators explicitly, or provide procedures that compute the `ngraph` *and equivalence classes.*

Table 1: Features of the spatial aggregation language.

the objects in the neighborhood graph. For example, *search* starts at any of a list of objects in the graph and moves from neighbor to neighbor, following some desired control strategy (e.g. depth-first search or breadth-first search) and finding paths satisfying some criteria. Interfaces to standard geometric and numerical libraries could further extend the capabilities of the language. Table 1 summarizes the main features of the spatial aggregation language.

Spatial Aggregation Language

We describe the spatial aggregation language and its prototype implementation in Scheme. The language consists of operators and abstract data types that a user can choose and instantiate for each spatial aggregate layer. A library available in the language contains basic implementations for each abstract data type and operator. Other component implementations can be built according to the defined specifications and added to the library if necessary.

We use a region-growing example from image analysis to illustrate the basic features of the language and how a simple program can be written for the task. A region is an area in an image whose pixels share a common property (Zucker 1976). The program takes as input a field — an image mapping pixel coordinates to brightness values — and produces a list of disjoint regions of pixels with similar brightness values. For example, given the input shown in Figure 11, the program could produce the regions shown in Figure 12. The pixel spatial objects encapsulate points and corresponding gray-scale values. The points belong to a sub-

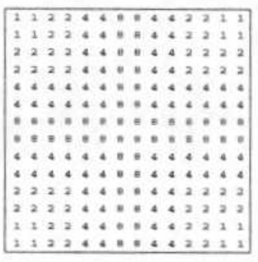

Figure 11: Input Image: A 14×14 array of pixels. Each pixel has a coordinate and a brightness value.

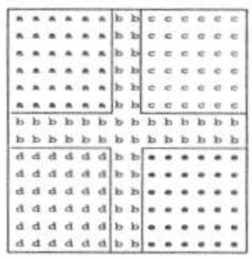

Figure 12: Output Image: five regions — a through e — consisting of pixels of similar brightness.

space of an Euclidean plane R^2, and the feature space comprises gray-scale values from the set $\{0, 1, ..., 255\}$ — a subspace of R^1. Pixels are neighbors by the standard 4-adjacency relation.

The field and N-graph abstract data types

Two abstract data types, the `field` and the `ngraph`, form the core of the spatial aggregation language. The operations of an abstract data type are separated into *primary operations* defining the basic capabilities of the data type in terms of its representation, and *secondary operations* (also called *capabilities*) providing extended functionality layered over the primary operations. The choice of primary and secondary operations is determined by the requirement of basic functionality and future language extension (Weide, Ogden, & Zweben 1991).

The abstract data types are highly parameterized. Each different implementation of an abstract data type provides an *instantiate* procedure that takes the parameters and returns a *facility* defining the primary operations. Parameters include functions specifying how to manipulate generic objects, and facilities for other data types used in the implementation. A facility is a function that, given a primary operation's name, returns the appropriate function. Each different implementation of a secondary operation also provides an *in-*

```
(define image-field-fac
  (field-array/instantiate '(256 256)
                           pixel/point))
```

Figure 13: Field instantiation for region growing

stantiate procedure that returns the secondary operation. While the basic abstract data types are highly parameterized, the library also provides data types whose parameters are *partially instantiated* with commonly-used values. The library allows users to choose among implementations trading efficiency with generality.

Field The first major component making up the spatial aggregation language is the `field` abstract data type. A `field` defines a metric space for the geometric descriptions of spatial objects, and can answer spatial queries. The interface provided by a `field` facility provides a function `create` to collect objects into a field, a function `domain` to return the objects defined in the field, and a function `near` to return objects within a given distance of a specified object. The syntax of these operators is omitted here due to the space limitation; interested readers are referred to (Bailey-Kellogg, Zhao, & Yip 1996) for details.

Many different spatial indices can serve as implementations for `field`. The current component library contains array and grid implementations, a k-d tree implementation, and a simple list implementation that compares objects pairwise. These components are generic, with instantiation parameters indicating how to extract geometric descriptions, how to measure distances, and so forth. Fields can also be defined intensionally or interpolated from sample data points; library components allow user-specified intensional and interpolation functions.

A 256 by 256 array field suffices for storing pixel values in the region-growing application, as shown in Figure 13.

N-graph The second main component of the spatial aggregation language is the `ngraph` abstract data type. An `ngraph` defines a neighborhood relation for a set of spatial objects, and can return the neighbors of any given object defined in the `ngraph`. The interface of an `ngraph` facility provides a function `create` to aggregate the objects of a field, and a function `neighbors` to return the neighbors of a specified object.

A wide variety of `ngraph` implementations support different neighborhood relations; each implementation uses a `field` facility indexing the objects to be aggregated. One implementation considers an object's neighbors to be all other objects within some distance in the field. Another implementation builds a minimal spanning tree of all objects and returns neighbors according to the minimal spanning tree. A third implementation builds a Delaunay triangulation with objects as vertices. Finally, a very generic implementation allows the user to specify a neighborhood-generating function. For example, a simple way to program an 4-adjacency neighborhood relation would be to provide a function that generates the four surrounding points and returns those defined in the field.

```
(define image-ngraph-fac
  (ngraph-near/instantiate image-field-fac 1))
```

Figure 14: Ngraph instantiation for region growing

```
(define image-ngraph
  (aggregate pixels image-ngraph-fac))
```

Figure 15: Aggregation of region-growing input

The code for the region-growing example, shown in Figure 14, defines a 4-adjacency neighborhood relation in terms of the nearness `ngraph`: return as a pixel's neighbors all pixels within a distance of one pixel.

Interface Operations

The spatial aggregation language provides the operations of Table 1 to manipulate primitive objects and neighborhood graphs.

Aggregate Aggregation groups objects of a field into a neighborhood graph. The function takes a field, along with an `ngraph` facility, and returns an `ngraph`:

aggregate: objects $*$ ngraph-fac $\rightarrow$ ngraph

This is simply syntactic sugar for the `ngraph` ADT's `create` primary operation.

Figure 15 shows the code to form a neighborhood graph for the region-growing application, with the objects from the image in Figure 11. The `pixels` list includes all pixels in the image.

Classify Classification is performed by an `ngraph` secondary operation. Given an `ngraph`, a list of objects to classify, and a threshold for feature similarity, this operation returns a list of equivalence classes (lists of objects):

classify: ngraph $*$ objects $*$ threshold $\rightarrow$ classes

The `objects` argument specifies the portion of an `ngraph` to classify, since the `ngraph` can be very large in space. There are several possible ways to determine what threshold to use for object similarity, each of which is supported by an implementation in the language library:

1. **Standard**: The easiest approach is to rely on a user-specified threshold. Objects are considered similar if they are neighbors in an `ngraph` and their difference according to the specified difference function does not exceed the threshold. An instantiation parameter specifies how object differences should be measured. Equivalence classes are found by taking the transitive closure of this similarity relation across neighboring nodes.[3]
2. **Splitting**: A more advanced method of determining a threshold is called *splitting* classification. To find a threshold by splitting, the system first classifies with a loose threshold. It then applies a user-supplied consistency check to each of the resulting equivalence classes. Classes that pass the consistency check are added to the results. Classes that fail the consistency check are reclassified with a tighter threshold. If the threshold bottoms out, failure is reported.
3. **Merging**: The third method for determining a threshold is called *merging* classification. To find a threshold by merging, the system starts classification using a tight threshold. It then examines the resulting classes and merges those that are similar. To determine which classes are similar, it views the classes as higher-level objects and aggregates them as another invocation of spatial aggregation.
4. **Stabilizing**: The final classifying method currently in the library is called *stabilizing* classification. Stabilizing classification compares the classifications yielded by a particular difference measure over a range of thresholds. It returns the classification that persists over the largest subrange of thresholds.

The code in Figure 16 takes the `ngraph` yielded by the code in Figure 15 and produces the classification of Figure 12. The instantiation uses a difference measure that compares pixel values and considers them equivalent if their difference doesn't exceed the specified threshold. Different instantiations could replace this classification method with one of the more powerful approaches.

[3]More precisely, two objects O_0 and O_n are considered equivalent if there exists a sequence of objects O_0, O_1, ..., O_n such that O_i and O_{i+1}, for i from 0 to $n-1$, are adjacent in `ngraph` and similar according to the similarity measure. The equivalence relation thus defined is a subset of the transitive closure of the neighborhood relation.

```
(define classify
  (classify-standard/instantiate
   image-ngraph-fac
   (lambda (n1 n2)
     (abs (- (pixel/value n1)
             (pixel/value n2))))))

(define classes
  (classify image-ngraph pixels *thresh*))
```

Figure 16: Simple classification of region-growing input

```
(define regions
  (redescribe classes region/create))
```

Figure 17: Redescribing equivalence classes as regions

Redescribe Redescribing maps equivalence classes to higher-level objects. The function takes a list of equivalence classes and a function to construct a higher-level object from a list of lower-level objects, and it returns a list of higher-level objects:

> **redescribe**: classes $*$ redescribe-function $\rightarrow$ objects

Given a function `region/create` to create a region object from a list of pixels, the code in Figure 17 performs the redescribing for the region-growing example.

Localize Localize inverts the work of redescribe. It takes a list of higher-level objects and a function to convert a higher-level object to its constituent lower-level objects, and returns a list of lists of lower-level objects:

> **localize**: objects $*$ localize-function $\rightarrow$ objects

Given a function `region/pixels` to return a list of points in a region, the code in Figure 18 generates the points forming the regions computed by the redescribe above.

Other operations N-graph secondary operations provide extended functionality. For example, a `search` operation returns paths through a neighborhood graph starting from any of a list of objects and satisfying specified goal predicate:

> **search**: ngraph $*$ objects $*$ goal-predicate $\rightarrow$ paths

Different implementations of this operation would provide depth-first control, breadth-first control, and so forth.

```
(define classes
  (localize regions region/pixels))
```

Figure 18: Localizing a region into its equivalence class

Finally, a variety of other operations, such as the standard Lisp operations map and filter, along with various geometric operations, manipulate primitive objects.

Application Domains of the Language

This section describes several application domains for the spatial aggregation language. In addition to image analysis and dynamical system analysis tasks, the spatial aggregation language is applicable to a wide range of other problem domains. Preliminary prototypical implementations have been developed for a number of these applications; others are under development.

Dynamical system analysis

Spatial aggregation generalizes KAM (Yip 1991), MAPS (Zhao 1994), and a number of other programs (Bradley 1992; Nishida & others 1991; Sacks 1991) for analyzing nonlinear dynamical systems. According to modern dynamical systems theory, the qualitative behaviors of a nonlinear dynamical system can be described by the geometric features in phase space. Once the appropriate metrics and equivalence relations are defined, the spatial aggregation language can naturally express the operations in analyzing dynamics in phase space, as the trajectory interpretation example has already illustrated.

Fluid flow motion analysis is another domain where spatial aggregation can be used to aggregate flow lines into coherent bundles (Yip 1995).

Mechanical mechanism analysis

A mechanical mechanism analysis determines the feasibility of a mechanism. HIPAIR is a program for this task (Joskowicz & Sacks 1991). A mechanism is often described in a configuration space where each dimension represents an independent degree of freedom. The configuration space consists of blocked space (impossible configurations) and free space (possible configurations). Spatial aggregation can be used determine the feasibility of a mechanism by clustering configurations into sign-invariant regions (free space or blocked space regions) with a Hausdorff distance measure between sets of points and then analyzing the connectedness of the free-space regions.

Image analysis

We have shown how the region growing operation can be coded in the spatial aggregation language. Other image analysis operations such as boundary tracing and segmentation can be likewise programmed. Mahoney (Mahoney 1995) developed a library of elementary image analysis operations in the style of Ullman's visual routines. An interesting open problem is to reimplement the Ullman's set of visual routines in the spatial aggregation language.

Other applications

Auditory scene analysis attempts to differentiate concurrent acoustic signals generated by distinct sources. An auditory scene is an image in which signals are separated by features such as tones, onset times and offset times. Spatial aggregation can partition an auditory scene into groups based on feature similarity and hence separate acoustic signals from distinct sources.

Data mining is another potential application domain. Data mining extracts regularities from a massive amount of data using correlation and generalization. The space of possible relationships among data items is very large and prohibits a brute-force search. By defining appropriate metric and equivalence relations for the data items, spatial aggregation might be able to exploit the spatial adjacencies among data and hence reduce the need to search.

Geographic information databases are another area in which the neighborhood of data items is already defined in a geographic space.

Language Experience

We have developed several small-scale application programs written in this language. Based on our experience, programming in the spatial aggregation language has several advantages:

1. The language allows a user to isolate what is important and express the important computational ideas in terms of the formation of equivalence classes and the transformation of neighborhood graphs, while hiding low-level implementation details. For example, the classify operator provides means for a user to specify and search for appropriate classification thresholds. The resulting programs are modular and concise.

2. The language provides field and N-graph data types for naturally representing physical objects in continuous domains. Field is a commonly used abstraction in science and engineering and hence facilitates the scientific and engineering applications of the language. N-graph serves as a common interface for

developing programs. The interface operators are identical for different layers of spatial aggregation.

3. For a given task, a user can craft a program by mixing and matching and specializing components from the library provided by the language. A user has fine control over efficiency and generality in the language implementation and can extend the language capability by adding additional component implementations. Specializing data types through partial instantiation can improve performance; so can a more efficient implementation of a component. For example, a k-d tree field facility that replaces a grid can improve the object indexing performance in manipulating non-uniformly distributed points.

The current implementation of the language is limited in a number of ways. We plan to incorporate additional types of components, provide additional component implementations, and improve computational efficiency of the implementation. Other goals include the implementation of lazy evaluation and incremental analysis and update for N-graphs. To apply the language to large-scale problems, we need to build interfaces to existing numerical and computational geometry libraries so that the language can tap the power of the existing software base.

Conclusion

We have described an implemented language that supports programming in the style of spatial aggregation for a number of small-scale applications ranging from dynamics interpretation to image analysis. The spatial aggregation language provides primitives — field, N-graph, and a small set of operators — and means of abstraction for building problem solvers that derive concise symbolic descriptions from analogue representations of physical phenomena. Our experience provides evidence that the language supports the development of modular programs at an appropriate level of abstraction.

A central problem in artificial intelligence is to understand and construct the mappings from analogue signals to symbols and back. Spatial aggregation achieves a descriptive economy for an analogue input field by successively forming equivalence classes of lower-level objects and transforming a multi-layer of spatial aggregates, and is a possible realization of the signal-to-symbol mapping. Many important research questions remain open: What class of scientific problems can be formulated and solved in the style of spatial aggregation? Is there biological evidence that the brain might be performing spatial aggregation? What are other styles of reasoning that might bridge the analogue signals with the symbols?

References

Bailey-Kellogg, C.; Zhao, F.; and Yip, K. 1996. Spatial aggregation: language and applications. Technical Report OSU-CISRC-1/96-TR04, Department of Computer and Information Science, The Ohio State University.

Bradley, E. 1992. Taming chaotic circuits. Technical Report AI-TR-1388, MIT Artificial Intelligence Lab.

Chandrasekaran, B., and Narayanan, N. 1990. Towards a theory of commonsense visual reasoning. In Nori, K., and Madhavan, C., eds., *Foundations of Software Technology and Theoretical Computer Science*. Springer.

Forbus, K.; Nielsen, P.; and Faltings, B. 1991. Qualitative spatial reasoning: the CLOCK project. *Artificial Intelligence* 51.

Joskowicz, L., and Sacks, E. 1991. Computational kinematics. *Artificial Intelligence* 51:381–416.

Mahoney, J. 1995. Signal-based figure/ground separation. Preprint.

Nishida, T., et al. 1991. Automated phase portrait analysis by integrating qualitative and quantitative analysis. In *Proceedings of AAAI.*

Sacks, E. 1991. Automatic analysis of one-parameter planar ordinary differential equations by intelligent numerical simulation. *Artificial Intelligence* 51:27–56.

Ullman, S. 1984. Visual routines. *Cognition* 18.

Weide, B.; Ogden, W.; and Zweben, S. 1991. Reusable software components. *Advances in Computers* 33:1–65.

Yip, K. M., and Zhao, F. 1996. Spatial aggregation: Theory and applications. *J. Artificial Intelligence Research.* To appear.

Yip, K. M. 1991. *KAM: A system for intelligently guiding numerical experimentation by computer*. MIT Press.

Yip, K. M. 1995. Reasoning about fluid motion: Finding structures. In *Proceedings of IJCAI.*

Zhao, F. 1994. Extracting and representing qualitative behaviors of complex systems in phase spaces. *Artificial Intelligence* 69(1-2):51–92.

Zucker, S. 1976. Region growing: childhood and adolescence. *Comput. Graphics Image Process.* 5.