Physics-Based Encapsulation in Embedded Software for Distributed Sensing and Control Applications

Feng Zhao, Chris Bailey-Kellogg, Markus P.J. Fromherz

Abstract—

Spatial Aggregation abstracts data arising from distributed embedded sensing and control applications as a set of so-called "spatio-temporal objects". Locality and continuity in the underlying physics of a problem domain give rise to spatially coherent and temporally contiguous objects in an appropriate metric space. Once parameterized by physical properties such as location, intensity (e.g., light, temperature, pressure), and motion (e.g., velocity), these objects can be aggregated and abstracted into more abstract descriptions. Applications are written as the creation and transformation of these abstract objects.

We illustrate how these objects naturally arise from applications such as distributed sensing and actuation and use an air-jet table system to demonstrate how such a physics-based encapsulation modularizes the design of sensing and control software. Unlike in traditional software design, where objects and operations are defined mathematically and possess a semantics independent of possible implementations, the objects in distributed embedded software are defined by the physics of the application, algorithmic considerations, task requirements, as well as optimization criteria. The air-jet table example demonstrates that the grouping and abstraction of actuation devices are determined by laws of motion, the type of force allocation algorithms used, and the desired performance of the controller; this encapsulation greatly simplifies the design and implementation of a force allocation algorithm for the system and improves software modularity. Based on our practical experiences in designing several massively distributed sensing and actuation systems, we present a set of recommendations for distributed embedded software modeling and design.

Keywords— **Physics-based encapsulation, embedded software design, distributed sensing and control.**

I. Introduction

The confluence of technological advances in MEMS sensors and actuators, wired and wireless networking, and embedded processing has enabled a new generation of massively distributed sensing and control systems. These systems, compared with their conventional siblings, are much more tightly coupled to the physical environments through direct local sensing, processing, and control. For example, an advanced airplane wing covered with an array of thousands of tiny MEMS flaps generates desired lift and steer vectors by regulating the motion of individual flaps to interact with the turbulent air flows around the wing [1]. These interactions with the physical environment occur at a scale much smaller and faster than those in conventional embedded systems found on, say, automotives. The overall effect on the physical environment by many of these massively distributed systems is a product of aggregating sensor data and actuation forces from individual sensors and actuators.

One such system that has been designed, fabricated, and demonstrated as part of the PARC Smart Matter projects is the air-jet table, a one foot by one foot experimental testbed made of 576 individually controllable air jets and 32,000 light sensing pixels [2], [3]. The air-jet table is used for controlling the motion of a macro object such as a sheet of paper by levitating and exerting a force on the object using air streams from the jets, just like an air-hockey table. The paper position and orientation are sensed by combining data from individual sensing pixels, and the motion of the paper sheet is controlled by distributing the desired overall force and torque to groups of individual jets. The notion of object abstraction is quite natural in this context. A virtual sensor is a group of sensors that detect aggregate phenomena such as an edge crossing, produced by combining data from the sensors in that group. A virtual jet is a group of jets that can be treated as one jet for the purpose of controlling the motion of the paper sheet. Different groupings of jets may be possible, depending on what control algorithms are used, and how fast and optimal the solution needs to be.

The implications of these considerations on embedded software design and modeling are profound. Cross-cutting concerns due to physics constraints often render conventional functional encapsulation impossible. Moreover, because of resource limitations in most embedded applications, the need to optimize the code for performance is often at odds with modularity and understandability. Another complication arises when objects that are seemingly far apart in the physical space are close according to the physics under consideration. For example, two distant air jets may be considered similar (or equivalent) if they apply the same amount of torque to a paper sheet. If the rotation of the sheet is the primary concern, then the two jets are close in the torque space. One needs to consider the physical, logical, as well as functional spaces when designing and optimizing embedded software.

There have been a number of approaches to modeling physical system behaviors using physics-inspired constructs (e.g., Modelica [4] and bond-graph based approach [5]). These approaches assume a lumped-parameter abstraction of a system and only model the temporal behaviors. Differential equations govern the behaviors of the system. In contrast, the dynamics of spatial interactions is a primary concern for spatially distributed embedded systems such as the air-jet table and the airplane wing. The modeling constraints are partial differential equations. Therefore,

Feng Zhao and Markus Fromherz are with Palo Alto Research Center (PARC), 3333 Coyote Hill Road, Palo Alto, CA 94304 USA (e-mail: fz@alum.mit.edu, fromherz@parc.com).

Chris Bailey-Kellogg is with Dept. of Computer Sciences, Purdue University, 1398 Computer Science Building, West Lafayette, IN 47907-1398 USA (e-mail: cbk@cs.purdue.edu).

the objects in distributed embedded software are defined by the physics of the problem, the geometry of the sensor and actuator layout, the communication patterns, the sensing and control algorithms being used, as well as the performance constraints on response time, resource consumption, and robustness to failure and degradation. By suitably encapsulating these properties into objects, cross-cutting constraints due to spatio-temporal dynamics become local properties of the objects and the modularity of the software can be greatly improved. By definition, these objects are partly defined by the physics, and hence the object abstraction should also aid in the understandability and debugging of the software. The interactions between the objects are not arbitrary, and are partly governed by the physical laws. Continuity and locality constraints apply.

Spatial Aggregation Language (SAL) is an object modeling tool for distributed sensing and control applications [6], [7]. In most physical problems, space and time are the two primary parameters. SAL defines spatio-temporal objects, objects that are spatially coherent and temporarily contiguous, as the physics-inspired modeling constructs. Objects are parameterized by the spatial and temporal parameters, as just discussed. In order to support more general object composition and transformation, we also endow spatio-temporal objects with additional parameterization such as intensity (e.g., light, temperature, pressure) or motion (e.g., velocity). This way, not only can objects interact according to spatial or temporal proximity, they can also be neighbors to each other in other suitable parameter spaces. For example, temperature contour objects on a weather map are sets of locations whose temperature measurements are equivalent in a temperature parameter space, while a "cold spot" is a collection of spatially contiguous locations whose temperatures are below a certain threshold. Once these objects are defined, they can be aggregated and abstracted into more abstract descriptions according to physical laws and task objectives. SAL generalizes these application-specific transformations to provide a set of generic operators. Applications are written as the creation and transformation of objects in SAL.

Spatio-temporal objects arise from locality and continuity of the underlying physical field of an application. A physical field tends to exhibit continuities of properties (such as light intensity or temperature or pressure). Consequently, the field can be divided into equivalence classes, i.e., open regions where a particular property varies in an approximately uniform way. With continuities we can achieve an economy of description by focusing on the open regions and their boundaries instead of the pointwise field. Higher-order continuities, i.e., continuities of properties defined on the open regions, can similarly be used to build more abstract spatio-temporal objects. SAL uses geometric constructs that have well-defined topological properties to capture this intuitive notion of continuity.

The abstraction of a physical field as a set of spatio-temporal objects presupposes the existence of continuity. From a methodological point of view, it is important to clearly identify the source of continuities in the field or equivalently in the physical system the field represents. The discovery of valid and general continuities in the physical system, as a sound mathematical basis for object modeling in embedded applications, is as much a scientific contribution as the subsequent computational use of them.

Understanding the unique characteristics of distributed embedded systems is key to embedded software development. For example, distributed systems are often communication limited. In the air-jet table system, sensor data must be aggregated before it can be used by the controller because of the physical limitation on the number of wires that can come out of the sensor array board. Power is a main consideration when the sensing and actuation nodes are untethered and must communicate through RF channels. This puts a severe limit on the communication bandwidth, sensing, and processing, and ultimately the network lifetime. Combining them with the physical embeddedness of the systems, these constraints present very interesting challenges to embedded software design.

The *contributions* of this paper can be summarized as follows. (1) This paper identifies the physics-based encapsulation as a powerful means of abstraction and composition for embedded software design. (2) It presents the SAL methodology to support the physics-based encapsulation. (3) It demonstrates how the physics-based encapsulation can lead to a simplified and modular design for a distributed embedded system, the air-jet table.

In the rest of this paper, we present the SAL methodology for object modeling, describe the SAL language constructs, and illustrate how SAL can be used to create and transform objects (Section II). We then present a case study of air-jet table design to illustrate how objects arise from the physics of the problem and how such considerations drastically simplify the design and modeling of the embedded control software for the system (Section III). We conclude the paper by drawing lessons learned from the SAL language design and the air-jet system prototyping and experimentation, and offer a few recommendations for future embedded software design (Section IV).

II. SAL: Spatial Aggregation Language

This section describes the design and implementation of the Spatial Aggregation Language for physics-based embedded software design. Section II-A introduces the underlying Spatial Aggregation framework. Section II-B then specifies the language in terms of its data types and operators, and Section II-C describes the language implementation as a C++ library and introduces an accompanying tool providing an interactive, interpreted programming environment for SAL. Section II-D illustrates the use of the language through an example program.

A. Spatial Aggregation Framework

Four key ideas of the Spatial Aggregation framework [6] are particularly important to embedded software design.

A.1 Field Ontology

Spatial Aggregation organizes computation around image-like representations of spatially distributed data. In the *field ontology*, 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 $\mathbb{R}^1$; a 2-D fluid field associates a velocity with each point, mapping from $\mathbb{R}^2$ to $\mathbb{R}^2$. Spatial Aggregation applications, such as fluid data interpretation [8] and phase space-based control [9], then uncover and manipulate structures in this representation through *imagistic reasoning* techniques [10].

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.

A.2 Multi-layer Spatial Aggregates

Spatial Aggregation uncovers structures at multiple levels of abstraction, with the structures uncovered at one level becoming the input to the structure-discovery process at the next level. For example, in a weather data analysis application [11], Spatial Aggregation could extract from pressure data the isobars, pressure cells, and pressure troughs. Such multi-layer structures arise from continuities in fields at multiple scales. Due to the 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 modular programs with 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.

A.3 Uniform Vocabulary

Spatial Aggregation provides a small set of uniform data types and concise operators for constructing the spatial aggregate hierarchy. The data types and operators make explicit use of domain-specific knowledge (see Fig. 1). Yip and Zhao [6] present a number of application programs, ranging from dynamical systems analysis to mechanical mechanism analysis, in terms of the same set of generic operators parameterized by different domain knowledge.

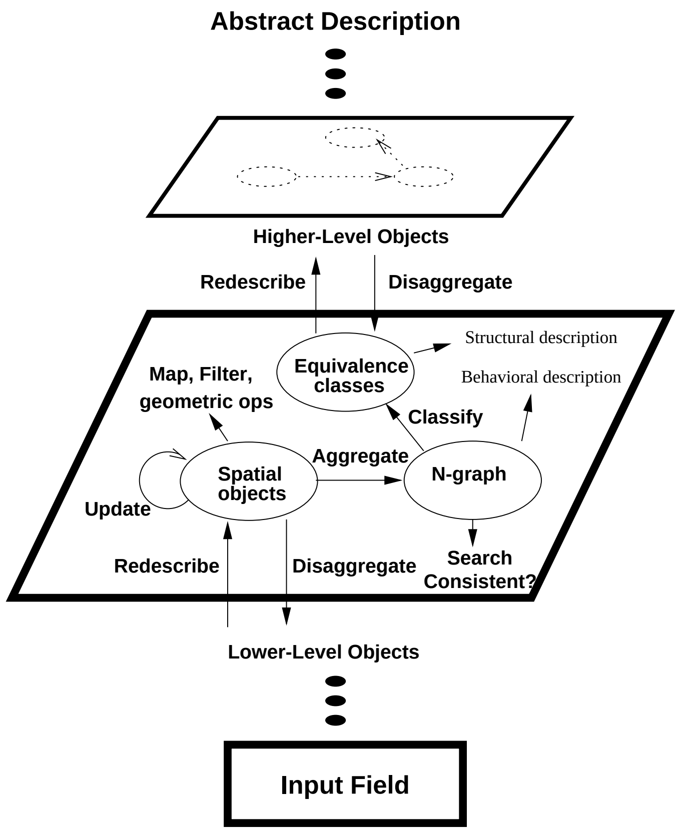


Fig. 1. Spatial Aggregation: objects are created, transformed, and abstracted using a set of generic operators.

The central data type of Spatial Aggregation, the *neighborhood graph*, is an explicit representation of an object adjacency relation. The definition of adjacency is domain-specific and depends on the metric properties of the input field. Common adjacency relations include Delaunay triangulations, minimal spanning trees, and uniform grids. The neighborhood graph serves as computational glue, localizing interactions between neighboring objects. The main Spatial Aggregation operators *aggregate* objects into neighborhood graphs satisfying an adjacency predicate, *classify* neighboring nodes into equivalences classes with respect to an equivalence predicate, *redescribe* equivalence classes into higher-level objects, and *localize* 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 application programs.

A.4 Structure-based Control Design

Spatial Aggregation control design applications utilize high-level, structural interpretations as the basis for determining low-level control actions. 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 [12]. 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 [13]. Controlling the mechanism requires moving its state through connected physically-possible regions in the configuration space. Finally, structural representations of thermal fields can be leveraged in the design of decentralized controls for temperature regulation [14]. In this influence-based model decomposition approach, bottom-up aggregation identifies coupling in a domain due to geometry and material properties; top-down disaggregation then leverages this abstract description in designing relatively independent decentralized control placement and parameters.

A.5 Example: Trajectory Bundling

As an example of applying Spatial Aggregation to specify a distributed analysis application, consider a simple trajectory bundler, following Yip's KAM program for analysis of dynamical systems [15]. 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). Fig. 2(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 flow directions.

The trajectory bundling application can be specified in Spatial Aggregation by the following steps:

1. *Points* to *trajectory curves*
 (a) Given input points (Fig. 2(a)).
 (b) *Aggregate* the points into a minimal spanning tree (Fig. 2(b)).
 (c) *Classify* connected points into the same equivalence class if the edge connecting them isn't too long relative to nearby edges (Fig. 2(c)).
 (d) *Redescribe* equivalence classes of consistent points as trajectory curves (Fig. 2(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 (Fig. 2(e)).
 (b) *Classify* connected curves into the same equivalence class if their flow direction is relatively similar (Fig. 2(f)).
 (c) *Redescribe* equivalence classes of consistent trajectory curves as trajectory bundles.

This example demonstrates the common computational structure of Spatial Aggregation programs: the same data types and operators are applicable at different abstraction levels (and in different tasks). The physical knowledge relevant for a task and abstraction level is encapsulated in parameters specifying notions of locality and similarity for the data types and operators. Section II-D further explores the SAL implementation of this application.

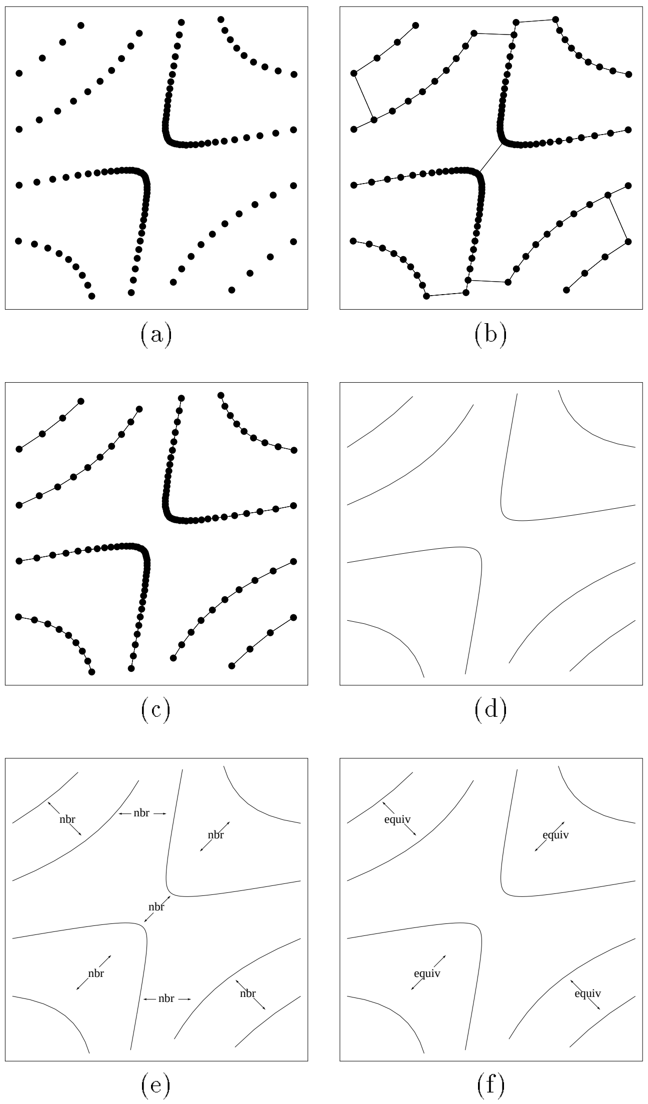


Fig. 2. Example steps in trajectory bundling application. (a) Input points. (b) Points aggregated into a minimal spanning tree. (c) Equivalence classes of points joined by short-enough edges. (d) Equivalence classes redescribed as trajectories. (e) Trajectories aggregated based on adjacencies of constituent points. (f) Equivalence classes of trajectories with similar-enough limit behavior.

B. Spatial Aggregation Language Definition

The Spatial Aggregation Language provides modeling constructs for data encapsulation commonly found in distributed embedded sensing and control applications. SAL components make explicit use of physical knowledge to uncover multi-layer aggregates of structures in physical fields. Tab. I categorizes the components into primitives, compounds, and means of abstraction [16], indicates the physical knowledge they exploit, and provides examples implemented by SAL. The rest of this section describes them in more detail, including the syntax for some of the main high-level operations beyond standard constructors and accessors. The space and time complexity depends on the implementations chosen; in a centralized implementation, for

Primitive
Spatial Objects O represent discrete objects in spatial data.
• *Cell Complexes X*: topological structure.
Ex: 0-d point, 1-d segment, 1-d nline, 2-d ngon, 2-d triangulation, 3-d nhedron.
• *Geometric Objects J*: properties with respect to *Metric Spaces*.
Ex: Coordinates, length, area, centroid, volume.

Compound
• *Spaces S* group objects; *Metric Spaces M* also encode distance metric and index objects.
Ex: Set, sequence, vector; array, k-d tree.
• *Fields F* map domains to features; encode continuity.
Ex: Scalar and vector fields.
• *Ngraphs G* map object to neighbors; encode locality.
Ex: Delaunay triangulation, minimal spanning tree, relative neighborhood graph, regular grid, k-nearest neighbors.
• *Classifiers E* partition objects into *equivalence classes*; encode similarity.
Ex: Cluster by best match, merge/split, transitive closure; compare by feature distance or bins in histogram.

Means of Abstraction
Abstractors A map compound objects at one level of abstraction to/from primitive objects at the next.
Ex: Bounding box, convex hull, curve from connected points in ngraph.

TABLE I
Components of the Spatial Aggregation Language.

example, a 2-d Delaunay triangulation can be computed in time $O(n \log n)$, while a fully-connected neighborhood graph requires time $O(n^2)$.

B.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. For example, spatial objects in a meteorology application could include points denoting sampled data, curves representing isobars connecting points of equal pressure, and regions indicating low pressure cells.

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

Distributed embedded systems have physical location — sensors and actuators measure and act upon local regions. This yields a low-level set of spatial objects in a SAL application. A key task is to leverage the physics of the embedding in order to uncover and utilize more complex, abstract spatial objects. The spatial object data type detailed here provides a uniform vocabulary and mechanism for composing, analyzing, and de-composing such spatial objects.

SAL distinguishes between the *topological structure* 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. In some cases only the structure is important; 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), yielding different geometric properties for the same structure.

B.1.a 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 SAL represents structures with *cell complexes* [17]. 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). 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. An object's *faces* include its proper faces, their proper faces, and so forth. A cell's *co-faces* are cells for which it is a face. A cell's *adjacent* faces are cells of the same dimension with which it shares a face (e.g., two contiguous line segments). Fig. 3 shows some examples of this hierarchy.

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

Definition 3 (Cell Complex) A *cell complex* is a collection of cells $\{c_1, c_2, \ldots, c_n\}$ 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.

Cell complexes are often constructed implicitly in the spatial aggregation process, as objects are uncovered in the data. Alternatively, a programmer can explicitly build a particular complex by specifying the appropriate relationships among its cells. The face/co-face/adjacency mechanism provides a uniform vocabulary and programming style for carrying out this process in different applications and at different abstraction levels. SAL provides a number of operators for manipulating cell complexes once built; Tab. II specifies the basic ones.

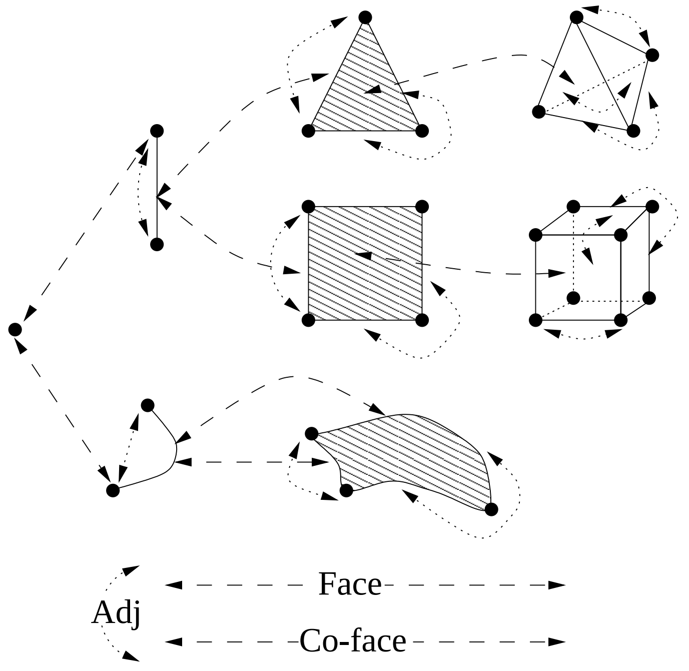


Fig. 3. Example cells. Cells of dimension n are the proper faces of cells of dimension $n+1$. Some face/co-face/adjacency relationships are shown.

• *dimension* : $X \to \mathbb{N}$
$x \mapsto \max_{k \in \mathbb{N}} \exists c \in x : c$ is a $k-$ cell.
• *adjacent_faces* : $X \times C \to S$
$(x,c) \mapsto \{c' \neq c \in x : (c \cap c' \neq \emptyset) \wedge \text{dimension}(c) = \text{dimension}(c')\}$.
• *cofaces* : $X \times C \to S$
$(x,c) \mapsto \{c' \neq c \in x : c \cap c' = c\}$.
• *faces* : $X \times C \to S$
$(x,c) \mapsto \{c' \neq c \in x : c \cap c' = c'\}$.

TABLE II
MAJOR CELL COMPLEX OPERATIONS.

B.1.b Geometric Properties. A spatial object's geometric properties depend on its type; for example, a point has coordinates, a segment has length, a curve has curvature at a specified point, and a polygon has area. These properties are defined with respect to a particular reference frame and *distance metric* such as the 2-D Euclidean metric $d(p,q) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2}$. SAL provides the *geometric object* and *metric space* data types to support these computations.

Definition 4 (Metric Space) A *metric space* for a collection of objects S defines a *metric* function $d : S \times S \to \mathbb{R}$ 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. $d(x,z) \leq d(x,y) + d(y,z)$.

Many metric spaces use *coordinates* to measure distances (e.g., Cartesian coordinates with Euclidean or Manhattan metric, or a polar coordinate system). Metric spaces for complex geometric objects can leverage their structure (e.g., a Hausdorff metric on vertices in two cell complexes, or a Euclidean metric on object centroids).

The metric space defining the geometric properties of a structure is known as the *embedding* space; for example, a square might be embedded in a 3-D space. 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. 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 and geometry.

Definition 5 (Geometric Object) A *geometric object* is a 4-tuple (o, m_e, m_l, p) where
• o is a spatial object for the structure.
• m_e is an embedding metric space.
• m_l is a local metric space.
• p is a set of object-specific geometric properties.

Instantiation of the object-specific geometric properties 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, either bottom-up (e.g., computing a triangle's area and angles based on the lengths of its sides) or top-down (e.g., using the angles and area of a triangle to define the lengths of its sides). Such properties are normally implicitly computed by an instance, upon specification of structure and embedding space or by transformation from 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, 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. As discussed regarding cell complexes, this uniform approach provides a mechanism for composing relationships among distributed objects.

B.2 Compound Objects

Compound objects in SAL 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.

B.2.a Space. A *space* is simply a collection of spatial objects, and serves as the basis for operating on a number of objects simultaneously, either independently or to cooperatively extract a global property.

Definition 6 (Space) A *space* is a set $\{o_1, o_2, \ldots o_n\}$ of spatial objects.

Spaces can be constructed either by explicitly adding or removing members, or by relation to other spaces (e.g., selecting a subspace satisfying a particular predicate or transforming objects in a space by translation and rotation). Spaces provide both element-wise operations, implic-

• *difference* : $S \times S \rightarrow S$
$(s_1, s_2) \mapsto s_1 - s_2$.
• *gather* : $S \times O \times (O * O \rightarrow O)) \rightarrow O$
$(s, i, \oplus) \mapsto i \oplus o_1 \oplus o_2 \oplus \ldots \oplus o_n$ where $s = \{o_1, o_2, \ldots, o_n\}$.
• *intersection* : $S \times S \rightarrow S$
$(s_1, s_2) \mapsto s_1 \cap s_2$.
• *map* : $S \times (O \rightarrow O) \rightarrow S$
$(s, c) \mapsto \{c(o) : o \in s\}$.
• *near* : $M \times J \times \mathbb{R} \rightarrow S$
$(m, j, r) \mapsto \{j' \in m : d[m](j, j') \leq r\}$.
• *nearest* : $M \times J \rightarrow J$
$(m, j) \mapsto \arg \min_{j' \in m} \{d[m](j, j')\}$.
• *select* : $S \times (O \rightarrow \{0, 1\}) \rightarrow S$
$(s, t) \mapsto \{o \in s : t(o)\}$.
• *union* : $S \times S \rightarrow S$
$(s_1, s_2) \mapsto s_1 \cup s_2$.

TABLE III
MAJOR SPACE OPERATIONS.

itly distributed over the collected objects, and collection-oriented operations manipulating the spaces themselves (Tab. III). For example, an operation to compute area could be distributed to all the polygon objects in some set; the global average could be gathered back. In modeling embedded systems, subspaces are often selected programmatically based on some criterion that unites them. Such a collection of sensors can cooperate to yield a higher-level picture; such a collection of actuators can cooperate to achieve a higher-level task.

Spaces that define metrics (refer again to the previous section), can 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 implemented by such metric spaces include grids, k-d trees, and Voronoi diagrams. This spatial indexing strategy is especially useful in conjunction with subspace selection, where all geometric objects are defined with respect to a common *base* metric space, and then a particular subset can be selected and indexed in a *derived* metric space, making available powerful query mechanisms without requiring repeated computation of geometric properties.

B.2.b Field. Spatial objects capture regions of physical space. Fields then associate features (e.g., temperatures, velocity vectors) with the regions (Fig. 4). The field is the cornerstone of Spatial Aggregation: applications seek to find structures in these image-like fields of spatially distributed data. Mathematically, a field is a mapping from one continuum (the *domain space*) to another (the *feature space*), for example $\mathbb{R}^n \rightarrow \mathbb{R}^m$. SAL uses metric spaces for both the domain and feature spaces.

Definition 7 (Field) A *field* is a mapping $m_d \rightarrow m_f$ from objects in a domain metric space m_d to objects in a feature metric space m_f.

Since the domain and features of a field are metric spaces, the field encapsulates a domain-specific notion of *continuity*. We expect objects close together in the domain to have features that are close together, where proximity depends on the metrics defined by the metric spaces. This point will be further discussed in the section on equivalence classes.

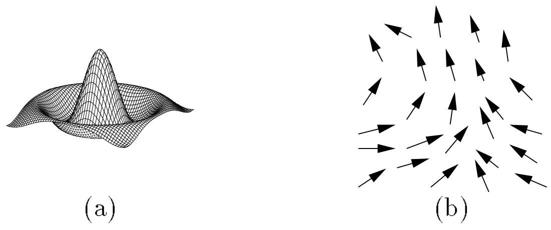


Fig. 4. A field associates objects and features. (a) Association of points and temperature values. (b) Association of points and wind direction vectors.

In SAL 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 with associated temperature intervals. Since the domain and feature spaces are discrete, a field can be instantiated as a set of pairs; for example, the $\mathbb{R}^3 \rightarrow \mathbb{R}^1$ temperature field could be implemented as a collection of ($\mathbb{R}^3$ point, $\mathbb{R}^1$ temperature) pairs, a representation particularly appropriate for distributed embedded systems, where a location naturally "owns" associated features (e.g., distributed sensors measuring temperature).

The field provides a uniform mechanism for querying and manipulating distributed spatial data (Tab. IV). Element-wise operations are implicitly distributed among the members of the field, for example to scale and add fields. More global field operations can, for example, select a subfield near a given element, interpolate implicit additional field values, and determine aggregate properties of a field (e.g., net velocity vector).

B.2.c 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 sensing applications, it is natural to work from local interpretations to more global ones. Similarly, to program distributed control applications, 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 (Fig. 5). A neighborhood graph forms adjacencies among objects based on a specified neighborhood relation (e.g., minimal spanning tree or k-nearest neighbors). The SAL *aggregate* operator serves as a constructor explicating such a predicate; explicit specification of neighbors is also supported.

Definition 8 (Ngraph) An *ngraph* is a pair (s, a) where
• s is a space.

• *combine* : $F \times F \times (J \times J \rightarrow J) \rightarrow F$
$(f_1, f_2, \oplus) \mapsto c = ((m_d[f_1] \cup m_d[f_2]) \rightarrow (m_f[f_1] \cup m_f[f_2]))$ such that

$$c(o) = \begin{cases} f_1(o) \oplus f_2(o) & \text{if } o \in m_d[f_1] \cap m_d[f_2] \\ f_1(o) & \text{if } o \in m_d[f_1] - m_d[f_2] \\ f_2(o) & \text{if } o \in m_d[f_2] - m_d[f_1] \end{cases}$$

• *gather* : $F \times J \times (J \times J \rightarrow J) \rightarrow J$
$(f, i, \oplus) \mapsto i \oplus (o_1, f(o_1)) \oplus (o_2, f(o_2)) \oplus \ldots \oplus (o_n, f(o_n))$ where $d[f] = \{o_1, o_2, \ldots, o_n\}$.
• *interpolate* : $F \times S \rightarrow F$
$(f, s) \mapsto f' = ((m_d[f] \cup s) \rightarrow (m_f[f] \cup m'_f))$ such that

$$f'(j) = \begin{cases} f(j) & \text{if } j \in m_d[f] \\ \text{some } j' \in m'_f & \text{else} \end{cases}$$

where m'_f is implementation-specific.
• *map* : $F \times (J \rightarrow J) \rightarrow F$
$(f, c) \mapsto f' = (m_d[f] \rightarrow m'_f)$ such that $f'(o) = c(f(o))$ and $m'_f = \{c(j) : j \in m_f[f]\}$.
• *select* : $F \times (J \times J \rightarrow \{0,1\}) \rightarrow F$
$(f, t) \mapsto (m'_d \rightarrow m'_f)$ where $m'_d = \{o \in m_d[f] : t(o, f(o))\}$ and $m'_f = \{f(o) : o \in m'_f\}$.
• *subfield* : $F \times S \rightarrow F$
$(f, s) \mapsto f' = (s \rightarrow m'_f)$ such that $f'(j) = f(j)$ for $j \in s$ and $m'_f = \{f(j) : j \in s\}$.

TABLE IV
Major field operations.

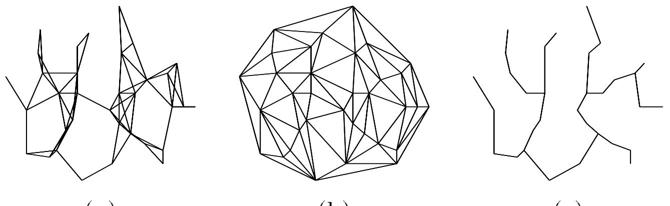


Fig. 5. Ngraphs localize computation with object adjacencies. (a) Neighbors within a fixed radius. (b) Neighbors by Delaunay triangulation. (c) Neighbors in minimal spanning tree.

• $a \subseteq s \times s$ is a set of *adjacencies* (o_1, o_2) between objects in s.

Note that while physical neighborhood relations are symmetric, the definition of ngraph allows directed adjacencies, which can be computationally useful for asymmetric neighborhood graphs such as k-nearest neighbors. Specializations of neighborhood graph construct additional structure besides adjacencies, such as a cell complex for a minimal spanning tree or Delaunay triangulation, or the *dual* graph for a mesh. Neighborhood graphs can also be specialized to take advantage of existing structure on the aggregated space; for example, a rectilinear *grid* can support queries about neighbors in a particular direction.

• *cell_complex* : $G \rightarrow X$
$g \mapsto s[g] \cup \{\text{segment}(o_1, o_2)$ for $(o_1, o_2) \in a[g]\}$ and implementation-specific higher-dimension cells.
• *closure* : $G \times \mathbb{N} \rightarrow G$
$(g, n) \mapsto (s[g], e^n)$ where

$$\begin{aligned} e^{i+1} &= e^i \cup \{(o_1, o_3) : (o_1, o_2) \in e^i \wedge (o_2, o_3) \in e^i\} \\ e^0 &= a[g] \end{aligned}$$

• directional_neighbor : $G \times J \times \mathbb{N} \times \{1, -1\} \rightarrow O$
$(g, j, a, d) \mapsto j' \in s[g]$ such that $(j, j') \in a[g] \wedge d(j' - j) \cdot e_a > 0$ where $e_a = a$th unit vector.
• *intersection* : $G \times G \rightarrow G$
$(g_1, g_2) \mapsto (s[g_1] \cup s[g_2], a[g_1] \cap a[g_2])\}$.
• *neighbors* : $G \times O \rightarrow S$
$(g, o) \mapsto \{o' \in s[g] : (o, o') \in a[g]\}$.
• *select* : $G \times (O \times O \rightarrow \{0,1\}) \rightarrow G$
$(g, t) \mapsto (s[g], \{a \in a[g] : t(o_1[a], o_2[a]) = 1\})$.
• *subgraph* : $G \times S \rightarrow G$
$(g, s) \mapsto (s, \{a \in a[g] : o_1[a] \in s \wedge o_2[a] \in s\})$.
• *union* : $G \times S \rightarrow G$
$(g, s) \mapsto (s[g_1] \cup a[g_2], a[g_1] \cup a[g_2]$.

TABLE V
Major neighborhood graph operations.

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 those 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 (Tab. V).

B.2.d Equivalence Classes. The main goal in Spatial Aggregation is to find and use structures — regions of uniformity — in distributed physical data. The SAL *classifier* mechanism 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* is a pair (s, p) where
• s is a space.
• $p \in 2^s$ partitions s into a set of subsets $\{s_1, s_2, \ldots, s_n\}$.

The SAL *classify* operator constructs a classifier when provided 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.

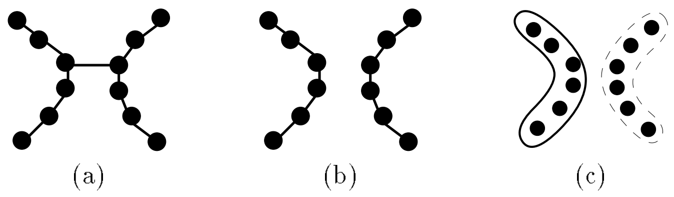


Fig. 6. Efficient classification process. (a) Localize comparisons with a minimal spanning tree. (b) Eliminate adjacencies for neighbors that are too far away. (c) Identify connected components as equivalence classes.

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 Fig. 6):
1. Localize comparisons with a neighborhood graph (e.g., MST).
2. Eliminate inconsistent adjacencies (e.g., too far away).
3. Find connected components in resulting graph (e.g., separate curve).

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. The application of this approach is particularly straightforward when combined with field data — the equivalence predicate is expressed in terms of the metrics of the domain and feature space (encoding a definition of continuity). This naturally leads to identification of iso-contours, "image" segmentation (where the generalized image is any physical field), and so forth.

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.

B.3 Means of Abstraction

The end result of equivalence class clustering is the identification of uniform sets of spatial objects, such as sensors yielding related information or actuators that can cooperate to achieve a task. Such a set can be composed into a single, more abstract, "first-class" object upon which the SAL mechanism can be brought to bear. 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, 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 to determine.

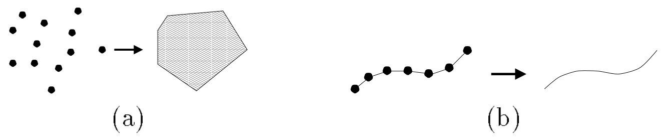


Fig. 7. Examples of abstraction. (a) A set of points is abstracted as a polygon bounded by the set's convex hull. (b) A set of points is abstracted as a curve based on adjacencies in the neighborhood graph.

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 (Fig. 7(a)), and a set of connected points can be abstracted as a curve (Fig. 7(b)); different implementations support such processes.

Definition 10 (Abstractor) An *abstractor* is a 4-tuple (s_l, p_l, s_h, u) where
- s_l is a space of lower-level objects.
- $p_l \in 2^{s_l}$ is a partitioning of s_l.
- s_h is a space of high-level objects.
- $u : p_l \rightarrow s_h$ 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. The SAL *redescribe* operator constructs an abstractor, based on an extensional or intentional specification of the mapping; its cache then supports the *localize* operator inverting the map.

C. 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 programming of modeling tasks 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.

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 span-

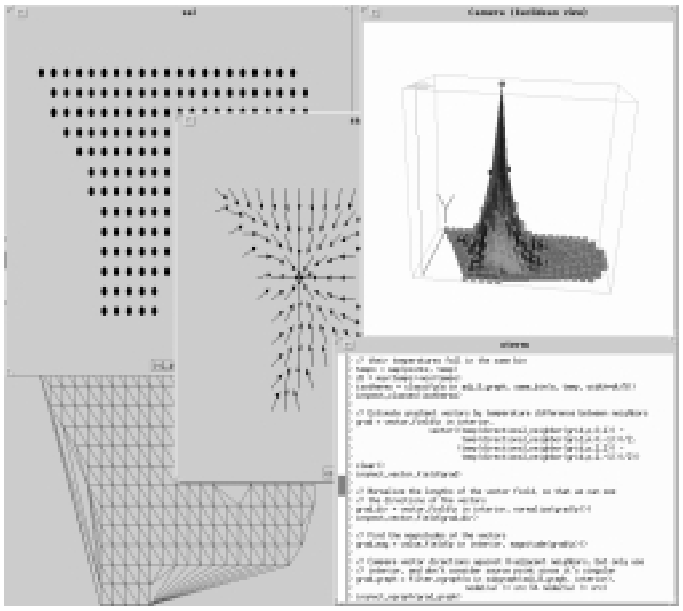


Fig. 8. The SAL interpreter in action: interactive code evaluation and graphical inspection of results.

ning tree, etc.) meeting that functionality in different ways. It also provides additional operators layered over the interface operators, supporting implementation-independent functionality. Tab. I lists some of the data type implementations currently provided by the library.

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. 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 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.

Fig. 8 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 programs by allowing the user to quickly and easily check the results of different instantiations of domain knowledge.

```
// (a) Read input points.
points = read_points(infile)

// (b) Aggregate into a Delaunay triangulation;
// find the minimal spanning tree.
point_trig = aggregate(points, make_mesh_delaunay())
point_mst = mst(point_trig)

// (c) 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_devs*sigma
}

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

TABLE VI

SAL code for first level of trajectory bundling application.

D. Example Program

This section demonstrates an actual SAL program run with the SAL interpreter discussed above. The trajectory bundling application, introduced in Section II-A.5, performs some of the work that the KAM program 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 (Tab. VI) starts by reading the *points* space of sample points. 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 (Tab. VII) 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* ab-

```
// (d) Redescribe point classes as trajectories.
points_to_traj =
  redescribe(classes(point_classifier),
             make_redescribe_op_path_nline(point_mst))
trajs = high_level_objects(points_to_traj)

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

// (f) 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 VII
SAL code for second level of trajectory bundling application.

stractor 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*). This substructure-based neighborhood graph, inspired by the strong/weak adjacency mechanism of Huang [11], 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, where correspondences are defined here by the structure of the point triangulation. *Traj_classifier* transitively follows this equivalence predicate through *traj_ngraph*.

E. Discussion

SAL provides a set of physics-based constructs for the modeling tasks associated with embedded systems design.

The physical knowledge about a domain (continuity, locality, linear superposability, etc.) is encapsulated as parameters (metrics, adjacency relations, equivalence predicates, etc.) to a uniform set of data types and operators. The data types are compositional — lower-level instances are combined in well-defined ways into more abstract instances, and the composition parallels the physics of the problem domain. Thus the programmer's efforts are focused on identifying high-level problem constraints, physical characteristics, what types of structures might arise, and what their relationships are. SAL then handles the details of instantiating a structural hierarchy for a particular problem instance. Section IV further discusses this programming style and when it is applicable. By encapsulating both physical and task-level knowledge as parameters of the SAL objects, SAL provides a natural framework to build modular embedded programs that directly read and act on physical signals. We next focus on a case study application of embedded system design and illustrate how considerations of physics of the domain and task requirements can yield useful abstractions of physical sensors and actuators for the software design.

III. Case Study: Design and Implementation of an Active Surface

As a case study, we present the design and implementation of an active surface developed at PARC and discuss it from the spatial aggregation point of view. This active surface is an air-jet table where multiple spatially distributed air jets act together to move an object such as a sheet of paper (Fig. 9a). While the object has only three degrees of freedom (x, y, angular position θ), we have tens of thousands of sensor pixels to reconstruct this information from, and hundreds of air jets to choose from to apply the desired forces. The problems of sensor fusion and force allocation are instances of the general problem of mapping between macro-level task information (e.g., object position and actuation) and the micro-level elements (e.g., sensor and actuators) of a highly distributed system. Clearly, to mediate between the macro level and the micro level, we will need to abstract sensors and actuators into more manageable descriptions at multiple levels of granularity so that efficient sensing and control algorithms can be implemented despite the scale mismatch.

The choices of sensor and actuator abstractions depend on the physics of the domain and other considerations. In this case, the physics of forces and object occlusion give rise to natural groups of actuators and sensors. In this section, we will present and discuss our choices in the implementation of algorithms for sensor fusion and actuation allocation. We will introduce the concept of virtual sensors as an abstraction of groups of physical or virtual sensors that collectively extract non-local information such as edge-crossing or sheet boundary, and the concept of virtual jets as an encapsulation of a group of individual jets that share certain common physical properties such as spatial proximity or co-linearity. The choices of virtual sensor and actuator objects can greatly influence the modularity and efficiency of the implementations of the sensing and control algorithms for this embedded application.

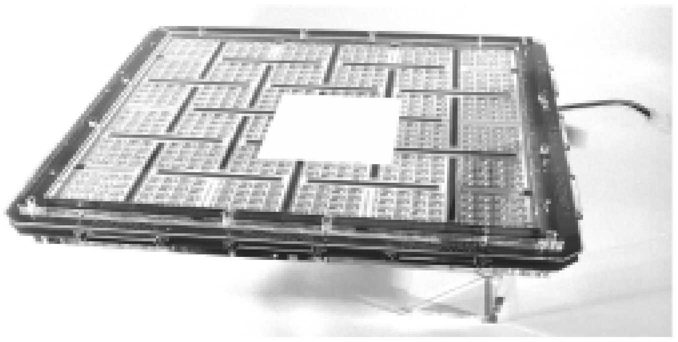
A. System Description

A.1 Hardware

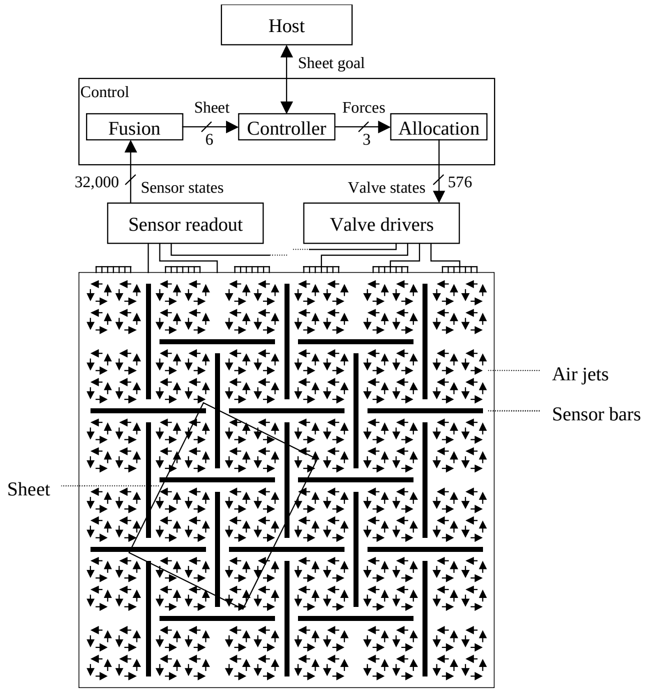
The air-jet system developed at PARC is a paper transport module that consists of an actuator and sensor board, driver electronics and software, and control software (Fig. 9b) [2], [18]. In this system, an array of 576 angled air jets is used to actuate a sheet of paper. The array consists of 36 (or 6×6) "tiles," and each tile consists of sixteen (or 4×4) jets. The angled air jets are arranged in four different directions (left, right, up, and down), which are distributed equally in the array. Individually computer-addressable electrostatic flap valves control the flow through each air jet. The air from the jets impinges on a sheet of paper floating roughly 2 mm above the air-jet array. Due to the viscous drag of the air against the paper, each jet applies a force to the paper largely confined in a small circular region around the impact point of the jet on the paper. To the first order, the forces of each jet add up as a linear super-position. Because the valves are either open or closed, each air jet is essentially a binary force actuator, and the entire array can be viewed as a distributed array of such actuators. The response time of the valves depends on the voltage applied to the flap valves and the supply pressure, but is typically on the order of 1-2 ms.

In addition, there are 25 CMOS image sensor bars located throughout the actuation surface, each of which consists of 1280 individual gray-scale sensors, for a total of 32,000 sensor pixels. The sensors are wired to detect edge positions of the paper. The sensor spacing is such that, for typical sheet sizes, there are always at least four points where a paper edge intersects the sensors. Thus, the x, y, and angular positions of the paper can be determined as the sheet moves over the array.

The desired paper state (x, y, θ, and their velocities) is obtained from either a joystick or pre-programmed trajectory from a file. The real-time control computations are performed in a 40 MHz Analog Devices SHARC DSP. The control rate is 500 Hz. Logging, monitoring, display, and control setup are performed by a host PC. We found this to be an ideal system to identify and solve control problems for large-scale distributed systems.



(a)



(b)

Fig. 9. Air-jet system: a) Photograph of 35 cm $\times$ 35 cm air-jet paper mover module: 16-element arrays are flap valves and associated jets, black bars are sensor arrays. b) System architecture and layout of the board.

A.2 Software

The implemented control system consists of the components as shown in Fig. 9b. However, some control operations that are trivial in small-scale systems dominate the control computations for large-scale systems. In particular, for large-scale systems, points in the control loop that require large dimensional transformations between high and low dimensions are of critical importance. For the present system, sensor fusion and actuation allocation are such critical steps.

The problem of *sensor fusion* involves converting the high-dimensional output from many sensors into a low-dimensional state representation within the control system loop time without losing relevant information. An estimate of the system state is determined by first aggregating the information from the individual pixels into transitions between dark and light, indicating sheet edge crossings. These are then further aggregated by fitting the points to the rectangle representing the sheet. The results of the sensor fusion step are the current estimated lateral and angular positions and velocities of the sheet.

In the *control* step, the estimated state is used to determine the necessary forces F_x and F_y and torque T_z (about the z-axis perpendicular to the sheet) to be applied to the sheet in order to move it to the desired state. We use a standard first-order proportional-derivative controller.

Finally, the problem of *actuation allocation* is to generate an optimal allocation for the air jets in order to produce the desired forces and torques $\mathbf{U} = (F_\mathrm{x}\ F_\mathrm{y}\ T_\mathrm{z})^\mathrm{T}$. Because of the enormous redundancy in actuators, one would like to use the additional degrees of freedom to attain other desirable goals. This problem can be represented as a constrained optimization problem, where the objective function expresses the desirability of various possible actuations and the constraints denote the required forces and torques.

Separating sheet control from force allocation has several advantages: (1) the dimensionality of control is 6 (positions and velocities) rather than 576, (2) the nonlinearity of binary valves and friction can be handled more easily in a low-dimensional control problem, (3) separation of control and allocation permits the investigation of the force allocation problem independently of particular control algorithms, and (4) changes in actuator configuration and operation do not require changes in the control algorithm.

B. Sensor Fusion

In the sensor fusion step, the output of 32,000 sensors must be condensed to yield the estimation for the state (x, y, θ, $\dot{x}$, $\dot{y}$, $\dot{\theta}$). The sensor fusion is accomplished using the following procedure.

B.1 Sensor Transformations

An important step in large-scale sensor fusion is discarding sensor outputs that do not contain useful information. Accordingly, the gray-level sensors are first binarized, after which the output of each of the 25 sensor strips is clocked out into a comparator that triggers when the sensor outputs change from light to dark or from dark to light. In other words, each strip reports only *crossings*, i.e., points where the strip crosses an edge of the paper. Most strips have zero or one crossing, but a strip near a corner of the paper may have two crossings. If an edge of the paper is almost aligned with a strip, the strip may give no reading or an erroneous reading. In this way, the gray values of 32,000 pixels are reduced to about 10 crossing points for each loop time.

The array of crossing points is then fit to a model of the paper: a rectangle of known dimensions or two pairs of parallel lines that are orthogonal to each other. The fitting procedure is essentially a least-squares fit that finds the best candidate rectangles. The fit most closely matching the expected position, based on the velocity and previous position, is selected as the current paper position. Low-pass filtered derivatives of the current position provide the translational and rotational velocity estimates. In the implemented system, we achieved a position accuracy of $25\mu m$ *rms* error, a result indicating effective sensor fusion. Sensor read-out and position calculation typically take about 0.3 ms.

B.2 Discussion

As explained, sensor fusion on the air-jet table proceeds in three main steps: binarization, fusion to edge crossings, and fusion to sheet position. The sensor pixels at the physical level output gray values. Sensors defined on top of these physical sensor pixels are all virtual sensors: a binary virtual sensor determines whether or not a sheet is present using the physical sensor output; a strip virtual sensor detects presence of paper sheet edges by combining binary detections from the sensors in a strip; and the board-level virtual sensor extracts the geometry of an entire sheet by fusing edge-crossings from the active strip sensors. Each virtual sensor is defined in terms of the sensors from the previous step.

Using SAL constructs, the sensor object can be parameterized as follows. A physical sensor pixel has point position and gray-level output. A binary sensor pixel has point position and binary detection. A strip sensor has line position and edge crossing position detection. Finally, a board-level sensor is characterized by the geometry of a patch of active strip sensors and paper sheet position detection. All sensors also have a confidence measure inverse proportional to the amount of noise in the sensor reading, e.g., the sharpness of the transition between light and dark in the strip virtual sensor. These sensor objects can be constructed recursively, starting from the physical sensor pixels, by defining equivalence in a parameter space of the lower-level objects and by abstracting the equivalence classes into aggregate objects. For example, active strip sensors, those that report at least one edge crossing, are combined using both physical proximity and equivalences in the numbers of edge crossings, to produce descriptions of boundary segments of a sheet. The formation of objects can be dynamically invoked, depending on the motion of the paper sheet. For example, the group of active strip sensors is defined by the paper motion.

The sensor fusion steps together perform an immense data reduction that is critical to ensure that the higher-level processing is not overwhelmed by non-essential information. Also, while these steps can be thought of as a series of transformations, they are implemented quite differently in the actual system. In particular, binarization and crossing computation are processed in hardware close to the sensor strips in order to avoid expensive communication channels. Also, the high-level optimization algorithm that reconstructs sheet position from edge crossings does not (have to) know the implementation of the actual sensors. This mapping from logical to physical implementation of the sensor fusion algorithm was a crucial part of the overall system design.

C. Force Allocation

The force allocation problem is defined as follows: given a model of the air-jet system and given the desired sheet forces, compute the appropriate state for each of the jet valves such that the air jets together deliver the desired forces. In the following, we first present a model for the air-jet system and then derive algorithms for performing the force allocation.

C.1 System Model

We present a generic model for a two-dimensional air-jet paper transport. We will also show how this model can be reused for different purposes, in particular for different levels in a hierarchical force allocation architecture. For example, we can model the entire transport component as a system whose "jets" are the tiles from which it is put together. Then, each tile can be modeled as a system that consists of the actual jets. Thus, each tile on the air-jet board is abstracted to a virtual jet for the next-higher level.

The air-jet paper transport consists of a set of N jets on a 2-d grid. A jet i ($i \in \{1, \ldots, N\}$) is represented by its

- position (x_i, y_i),
- force domains $d_{\mathrm{x}i}$ and $d_{\mathrm{y}i}$, and
- forces $f_{\mathrm{x}i}$ and $f_{\mathrm{y}i}$ ($f_{\mathrm{x}i} \in d_{\mathrm{x}i}$, $f_{\mathrm{y}i} \in d_{\mathrm{y}i}$).

The position (x_i, y_i) is given relative to the center of mass of the sheet. In this model, we represent only the jets under the sheet at the time of interest.

A force domain d may be a continuous interval $d = [f_{\min}, f_{\max}]$ or a discrete set of possible values $d = \{f_{\min}, f_{\min} + f_{\mathrm{inc}}, f_{\min} + 2f_{\mathrm{inc}}, \ldots, f_{\max}\}$ for some increment value f_{inc}. An actual jet in our system can only apply an x-force or a y-force in either positive or negative direction, and the jet can only be on or off (because of the binary valve). Thus, $d_{\mathrm{x}i} = \{0, f_{\max}\}$ for a positive x-jet and $\{-f_{\max}, 0\}$ for a negative x-jet; $d_{\mathrm{y}i} = \{0\}$ for both. The equivalent holds for y-jets. For this paper, we assume that jet valves can switch state instantaneously. In our current system, we have measured this time to be on the order of the control cycle, namely 1-2 ms, which has proven to be negligible relative to the sheet motion.

In general, a group of jets, e.g., a tile of 16 jets (cf. Fig. 9b), may be combined to form a virtual jet. Such a virtual jet may contain both x and y-jets and thus have both x and y-force components. For example, the maximum force for a virtual jet with n x-jets and n y-jets, with equal numbers of jets in positive and negative directions, is $f_{\mathrm{MAX}} = (n/2)f_{\max}$ in either direction. Thus, the domains for such a virtual jet are

$$d_{\mathrm{x}i} = d_{\mathrm{y}i} = \{-f_{\mathrm{MAX}}, -f_{\mathrm{MAX}} + f_{\max}, \ldots, f_{\mathrm{MAX}}\} \qquad (1)$$

A virtual jet's position (x_i, y_i) may be given as the centroid of the bounding box or the average position of its jets. The jets in a virtual jet may themselves be virtual jets.

Together, the jets in a virtual jet or an entire system deliver an x-force F_{x}, a y-force F_{y}, and a z-torque T_{z} to the sheet. A virtual jet i contributes additively with forces $f_{\mathrm{x}i}$ and $f_{\mathrm{y}i}$ to the system's x and y-forces, respectively. (This linear super-position has been verified by self-identification [3].) Furthermore, force $f_{\mathrm{x}i}$ contributes with $-y_i f_{\mathrm{x}i}$ to the z-torque (about the sheet's center of mass), and force $f_{\mathrm{y}i}$ contributes with $x_i f_{\mathrm{y}i}$. In other words, the z-torque delivered by virtual jet i is $t_{\mathrm{z}i} = x_i f_{\mathrm{y}i} - y_i f_{\mathrm{x}i}$. The same holds for actual jets, except that they deliver only either an x-force or a y-force. The total x-force F_{x}, y-force F_{y}, and z-torque T_{z} acting on the sheet are determined according to the following linear summation model:

$$\begin{aligned} F_{\mathrm{x}} &= \textstyle\sum_{i=1}^{N} f_{\mathrm{x}i} \\ F_{\mathrm{y}} &= \textstyle\sum_{i=1}^{N} f_{\mathrm{y}i} \\ T_{\mathrm{z}} &= \textstyle\sum_{i=1}^{N} x_i f_{\mathrm{y}i} - \sum_{i=1}^{N} y_i f_{\mathrm{x}i} \end{aligned} \qquad (2)$$

C.2 Allocation Algorithm

As mentioned, a top-level controller computes the required forces and torque $\mathbf{U} = (F_{\mathrm{x}}\ F_{\mathrm{y}}\ T_{\mathrm{z}})^{\mathrm{T}}$. The force allocation algorithm then determines which jets should be activated in order to deliver the required forces.

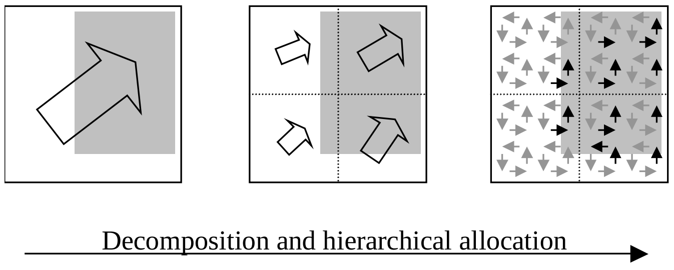


Fig. 10. Decomposition of a force allocation problem into sub-problems on an active surface

Because the number of actuators greatly exceeds the dimensionality of the control command $\mathbf{U}$, there are many ways of assigning the actuation to the jets. As already mentioned, the force allocation problem can be viewed as a constrained optimization problem, where the constraints are both the jets' domain constraints and the relations between individual jets and total delivered forces (Eq. (2)). This optimization problem is solved at each control time step in order to determine which jets should be activated. Precise generation of the required action is not usually possible because the actuation is discrete. Furthermore, such an optimization problem is in general NP-hard, i.e., actuation assignment scales poorly with the number of actuators. On the other hand, for very large numbers, the actuation approaches a continuum, and the problem becomes easy to solve approximately. Such a solution is described here.

The basic idea of our force allocation approach is the following. For small numbers of jets, an optimal assignment of actuation can be obtained by exhaustive search, a discrete optimization solver, or a lookup table of pre-computed solutions. If the number of actuators is large, the problem is decomposed into smaller sub-problems in a near optimal manner using continuous solutions as approximations. Each sub-problem may be further decomposed into yet smaller sub-problems, or, if sufficiently small, the sub-problem is solved optimally as just indicated (cf. Fig. 10).

In our application, decomposition into sub-problems consists of aggregating the jets into virtual jets (groups of jets) and then assigning responsibility to produce the required forces to each group. Because of the discrete nature of the actuation, the solution is not guaranteed to be globally optimal.

C.2.a Parameterized Allocation Function. Consider the problem of generating an optimal allocation of desired force $\mathbf{U}$ given a number of virtual jets (consisting of groups of jets or single jets). Assume that there are N virtual jets with jets in x and y directions. Each virtual jet is located at a position (x_i, y_i) ($i = 1, \ldots, N$) relative to the sheet's center of mass, i.e., the virtual jet applies its force at position (x_i, y_i). Consider the objective that jets should not work against each other while producing $\mathbf{U}$ (i.e., total actuation should be minimal). Hence, a possible constrained optimization problem capturing this desired behavior is

$$\begin{array}{rl} \min_{f_{\mathrm{x}i}, f_{\mathrm{y}i}} & \frac{1}{2}\sum_{i=1}^{N} \frac{f_{\mathrm{x}i}^2}{w_{\mathrm{x}i}^2} + \frac{1}{2}\sum_{i=1}^{N} \frac{f_{\mathrm{y}i}^2}{w_{\mathrm{y}i}^2} \\ \text{s.t.} & F_{\mathrm{x}} = \sum_{i=1}^{N} f_{\mathrm{x}i} \\ & F_{\mathrm{y}} = \sum_{i=1}^{N} f_{\mathrm{y}i} \\ & T_{\mathrm{z}} = \sum_{i=1}^{N} x_i f_{\mathrm{y}i} - \sum_{i=1}^{N} y_i f_{\mathrm{x}i} \end{array} \qquad (3)$$

where $f_i = (f_{\mathrm{x}i}\ f_{\mathrm{y}i})$ are the allocated x and y forces for virtual jet i, and $w_i = (w_{\mathrm{x}i}\ w_{\mathrm{y}i})$ are the weighting factors for each virtual jet's contribution to the x and y force. A large weighting factor causes the virtual jet to assume a greater role in meeting the constraints for forces and torque and a smaller role in minimizing the objective function. This objective function causes each virtual jet to minimize its actuation, while still providing the forces needed by the control.

If activation levels are continuous, the solution to this optimization problem using Lagrange multipliers is given by allocation functions

$$\begin{array}{l} f_{\mathrm{x}i} = w_{\mathrm{x}i}\left(\frac{F_{\mathrm{x}}}{\sum_{j=1}^{N} w_{x_j}} + \frac{T_{\mathrm{z}} - F_{\mathrm{y}}\overline{x} + F_{\mathrm{x}}\overline{y}}{\sigma_{\mathrm{x}}^2 \sum_{j=1}^{N} w_{y_j} + \sigma_{\mathrm{y}}^2 \sum_{j=1}^{N} w_{x_j}}(\overline{y} - y_i)\right) \\ f_{\mathrm{y}i} = w_{\mathrm{y}i}\left(\frac{F_{\mathrm{y}}}{\sum_{j=1}^{N} w_{y_j}} + \frac{T_{\mathrm{z}} - F_{\mathrm{y}}\overline{x} + F_{\mathrm{x}}\overline{y}}{\sigma_{\mathrm{x}}^2 \sum_{j=1}^{N} w_{y_j} + \sigma_{\mathrm{y}}^2 \sum_{j=1}^{N} w_{x_j}}(\overline{x} - x_i)\right) \\ t_{\mathrm{z}i} = 0 \end{array} \qquad (4)$$

for $i = 1, \ldots, N$, where

$$\begin{array}{ll} \overline{x} = \frac{\sum_{i=1}^{N} w_{\mathrm{y}i} x_i}{\sum_{i=1}^{N} w_{\mathrm{y}i}}, & \overline{y} = \frac{\sum_{i=1}^{N} w_{\mathrm{x}i} y_i}{\sum_{i=1}^{N} w_{\mathrm{x}i}} \\ \overline{x^2} = \frac{\sum_{i=1}^{N} w_{\mathrm{y}i} x_i^2}{\sum_{i=1}^{N} w_{\mathrm{y}i}}, & \overline{y^2} = \frac{\sum_{i=1}^{N} w_{\mathrm{x}i} y_i^2}{\sum_{i=1}^{N} w_{\mathrm{x}i}} \\ \sigma_{\mathrm{x}}^2 = \overline{x^2} - \overline{x}^2, & \sigma_{\mathrm{y}}^2 = \overline{y^2} - \overline{y}^2 \end{array} \qquad (5)$$

The quantities $\overline{x}$,$\overline{y}$ are the weighted average positions of all the modules, and σ_{x},σ_{y} are the weighted standard deviations of all the x and y virtual jet positions about this average. The torque $t_{\mathrm{z}i}$ is zero with respect to (x_i, y_i). The interested reader is referred to [19] for further details.

The allocation functions provide a continuous, closed-form solution to the optimization problem. They provide an optimal solution if the domains of the jets are continuous. When implemented with discrete actuators, the optimal solution is approximated with a hierarchical decomposition as the number of jets in each virtual jet becomes large (and thus has an almost continuous domain). The implementation with discrete actuators will be revisited later.

C.2.b Instantiating the Allocation Function. The solution functions to the constrained optimization problem can be instantiated as needed for particular systems. For example, for non-hierarchical (decentralized) allocation to the actual jets, each "virtual" jet consists of a single jet and is weighted equally. More interesting is two-level hierarchical allocation, where the system is decomposed into virtual jets consisting of individual jets. Relative weights for x and y-forces at the top level can be used to shift assignment of actuation depending the number of x and y-jets inside the virtual jets. Allocation within each virtual jet is akin to non-hierarchical allocation to the jets inside.

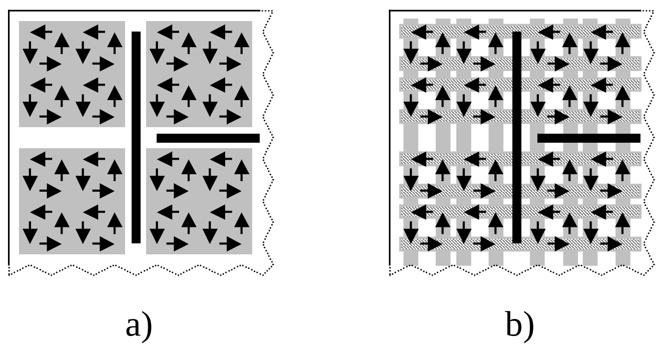


Fig. 11. Aggregation heuristics applied to the air-jet table (excerpts): a) neighborhoods in geometric space (tiles); b) neighborhoods in geometric and force space (rows and columns)

We are free to choose how to aggregate jets into virtual jets. One obvious heuristic is to follow the geometric layout, e.g., each tile of jets becomes a virtual jet (Fig. 11a). The advantage of this heuristic is that a modular structure of the system leads to a corresponding modular structure of the allocation algorithm. For an 8 1/2 by 11 inch sheet, which covers about 360 jets in our system, this heuristic decomposes the area under the sheet into about 40 virtual jets with up to 16 actual jets each. A virtual jet entirely covered by the sheet contains 16 actual jets, with 4 jets for each direction. Even with discrete valves, such a virtual jet provides near-continuous force domains for $f_{\mathrm{x}i}$ and $f_{\mathrm{y}i}$.

Another heuristic is to aggregate jets in neighborhoods of a different parameter space, namely that of force directions and location along just one of the axes. This heuristic is inspired by the linear superposition model (Eq. (2)). We observe, for example, that all x-jets with the same y position are interchangeable with respect to the constraints in Eq. (3). In fact, if our system consisted only of N x-jets with the same y position, the instantiation of the allocation functions in Eq. (4) would simply be

$$f_{\mathrm{x}i} = \frac{F_{\mathrm{x}}}{N}, f_{\mathrm{y}i} = 0, t_{\mathrm{z}i} = 0 \quad (i = 1, \ldots, N) \qquad (6)$$

For discrete actuators, one can simply calculate the number of necessary jets as $F_{\mathrm{x}}/f_{\mathrm{max}}$ rounded to the nearest integer and then open that many jets in whatever pattern seems suitable (e.g., inside-out, outside-in, or evenly distributed).

Generally, rows of x-jets with the same y position and columns of y-jets with the same x position lead to particularly simple allocation functions within a row or column. Therefore, we have implemented an aggregation of jets where the N virtual jets are divided into "x modules," each with only x-directed jets with a common y position y_i, and "y modules," each with only y-directed jets with a common x position x_i (Fig. 11b). Now, in instantiating the allocation functions, allocation to these virtual jets has to compute only either $f_{\mathrm{x}i}$ or $f_{\mathrm{y}i}$ for the x and y modules, respectively, and allocation within a virtual jet follows Eq. (6) for x modules and its equivalent for y modules.

Thus, the decomposition at one level can greatly simplify the allocation at another level. For an 8 1/2 by 11 inch sheet, this heuristic decomposes the area under the sheet into about 50 virtual jets with up to about 12 actual jets each.

C.3 Implications of Physics-based Encapsulation

Given the allocation functions and an aggregation of the actuators into progressively larger virtual air jets, the force allocation algorithm recursively assigns force to each (virtual) jet according to the allocation functions in a top-down manner. This algorithm also has the property that the allocation within one object — whether an actual or virtual air jet — does not depend on the allocation within another object at the same level. In other words, allocations within objects at any given level are completely decoupled from each other and thus can be parallelized and distributed among multiple processors. Furthermore, this possible distribution of computation is at the same scale as the actuation. This property is of obvious interest for distributed embedded systems with very large numbers of actuators.

As indicated, the parameters of the objects together with the definition of the allocation problem (Eq. (3)) help determine suitable aggregations for force allocation. It is instructive to compare the impact of hierarchical decomposition and the two aggregation heuristics (Fig. 11) on compute time and solution quality. Accordingly, we evaluated four algorithms: 1) centralized optimal allocation; 2) decentralized (non-hierarchical) allocation using the allocation functions; 3) two-level hierarchical allocation using the allocation functions at both levels; and 4) two-level hierarchical allocation using the allocation functions at the top level and optimal allocation at the lower level. For now, we assume that jets in the hierarchical algorithms are aggregated according to the purely geometric heuristic, i.e., into tiles (Fig. 11a), which means that the scaling complexity within a virtual jet is the same as that of the entire system. Also, allocation within virtual jets is assumed to be done in parallel.

As an example, we review the scaling of allocation error and computation time as the number of jets ranges from 10 to 100. (See [19] for full details as well as other evaluations.) While we cannot compute the optimal allocation for more than 10 jets, we have determined that the *rms* error is given by $0.4/N$ and computation time scales as 0.04×10^N for N jets. Thus, we compare the scaling properties of the other, near-optimal allocation algorithms and use the scaling laws for the optimal allocation. The computation times for the evaluated algorithms as a function of the number of jets are shown in Fig. 12. The hierarchical allocation algorithms exhibit a very slow increase with the number of jets compared with the centralized and decentralized algorithms, confirming the expectation that hierarchical allocations exhibit good scaling properties. The time of (sub-optimal) decentralized allocation (Alg. 2) increases as N, a clear disadvantage when compared with hierarchical al-

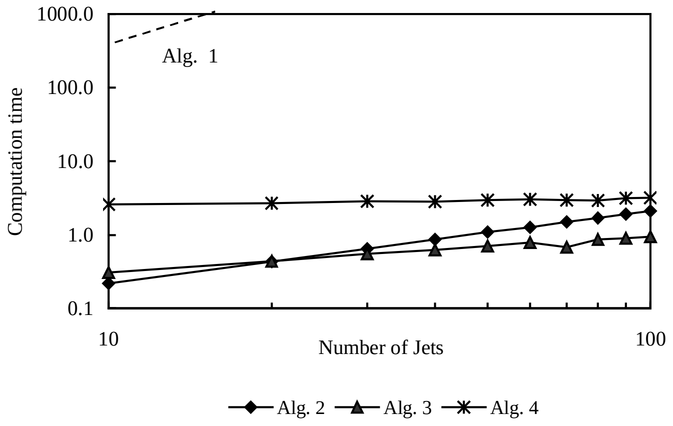


Fig. 12. Scaling of computation time of algorithms 1 through 4

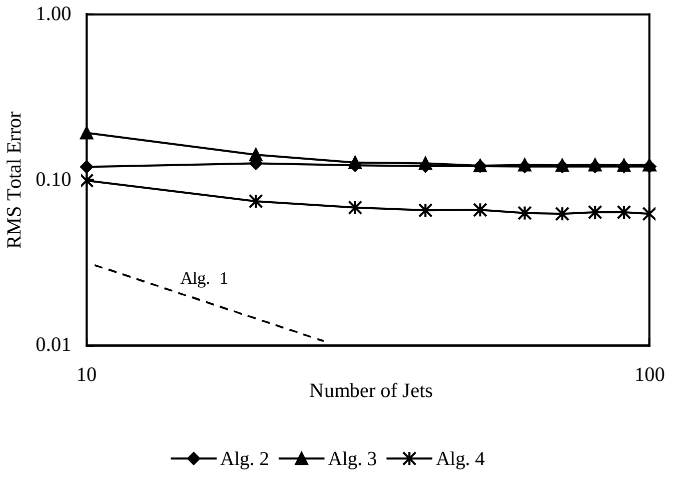


Fig. 13. Scaling of error of algorithms 1 through 4

location. The constant penalty value for Alg. 4 compared with Alg. 3 reflects the overhead required to compute an optimal allocation for a small number of jets. In fact, at $N = 10$, the computation time is completely dominated by the optimal allocation inside the virtual jets.

The *rms* error for the evaluated algorithms as a function of the number of jets is shown in Fig. 13. As is evident, the error does not decrease significantly as the number of jets increases, even with optimal allocation to virtual jets (algorithm 4). This behavior is a consequence of the fact that errors from individual virtual and actual jet allocations are not compensated by other allocations. With optimal allocation inside virtual jets, at least the jets within are able to compensate for each other's errors.

As mentioned, these time-complexity results assume tile-based aggregation in the hierarchical algorithms. While the scaling of the error is independent of the choice of aggregation, Alg. 4 carries a constant computation penalty over Alg. 3 for the optimization in the virtual jets. Now, as has already been indicated, this overhead is effectively removed by choosing the alternative aggregation into rows and columns, taking both geometric location and force direction into account. With this aggregation, optimal allocation within a virtual jet can be computed in constant time (cf. Eq. (6)). The resulting algorithm has the low time complexity of Alg. 3 (Fig. 12) and the low errors of Alg. 4 (Fig. 13).

C.4 Discussion

We can summarize these results as follows (see [19] for details). As expected, there is a trade-off between error and computation time for the force allocation problem for large numbers of actuators. Centralized optimal allocation (Alg. 1), while producing the smallest errors, scales poorly with increasing numbers of jets. In fact, unless optimal allocation can be pre-computed or computed easily (as in the row/column decomposition), the computation time scales as $2N$ for N jets. On the other hand, decentralized allocation (Alg. 2), or simply breaking the problem into virtual jets that are not allocated optimally (Alg. 3), results in relatively large allocation errors. In fact, the dominating *rms* error is on the order of one half the maximum total force. The best compromise is a hierarchical decomposition into virtual jets with optimal allocation within them (Alg. 4), especially if the local optimization can be computed in constant time (through pre-computation for small virtual jets or row/column aggregation). We also found a possible trade-off between the size and number of virtual jets in order to decrease the error while keeping computation time within acceptable bounds.

The general idea behind the hybrid hierarchical-optimal algorithms presented here should be transferable to other systems with large numbers of uniform actuators: aggregate the actuators such that actuator groups are large enough to be able to approximate the desired actuation within a given error, but not so large that the actuation allocation couldn't be computed in the available time frame.

This hierarchical allocation algorithm has been applied to the actual MEMS system and found to work well [2]. The algorithm performs the allocation in less than 0.4 ms for row/column aggregation on a single Analog Devices SHARC 40MHz DSP using Alg. 4. As mentioned, an 8 1/2 by 11 inch sheet in our system covers about $N = 360$ jets, and row/column aggregation results in about $M =$ 40 to 50 virtual jets with an average of 7 to 9 jets and a maximum of about 12 jets inside (all depending on sheet position). The average force allocation error is $(M/2)f_{\max}$, which is about 20-25 times $f_{\max}$. The resulting position error has been $30\mu m$ *rms* and 5-10 radian *rms* angle error.

As with the sensor fusion algorithm, the logical architecture of the hierarchical allocation algorithm has to be mapped to an actual, physical architecture. While the current system runs the algorithm on a single processor, the progressive aggregation of jets underlying the algorithm suggests a distributed implementation with multiple processors and communication embedded accordingly into the surface of the air-jet table. The principle is the same as with the mapping of the sensor algorithm: to delegate as much of the processing for this large-scale problem to lower levels, close to the sensors and actuators.

IV. Physics-based Modeling and Design for Embedded Software

A. Physics-based Encapsulation

We have described how SAL can support modeling tasks for embedded software design by providing a set of geometric constructs with well-defined properties. The creation and composition of these geometric objects are governed by the physics of spatio-temporal phenomena. This style of modeling can be summarized as follows:
Encapsulate knowledge of locality, continuity, and spatio-temporal scales in spatially distributed physical data as parameterizable multi-layer spatial-temporal structures.

This section discusses what that means and how to approach a modeling and design task in this style. In particular, it examines the types of applications for which physics-based encapsulation is appropriate and how this approach impacts the design and implementation of embedded software.

B. When Is Physics-based Encapsulation Appropriate?

Recall that in Spatial Aggregation computations are structured around perception-like operations on image-like representations of data. A key part in using physics-based encapsulation as a modeling methodology is to adopt the "imagistic stance" by encoding a task in terms of geometric and topological structures. For example, in the air-jet system design, each sensor pixel at the physical layer is viewed as a point in the configuration space of the air-jet board. Aggregate objects (i.e., virtual sensors) arise from grouping the pixel sensors into strip sensors and then edge boundary sensors, as defined by the sensing physics and task requirement of motion tracking. Likewise, pointwise individual air jets are grouped into virtual jets according to how the force and torque produced by the jets affect the motion of the sheet.

Adopting this stance yields the following lower/higher-level characteristics for physics-based modeling problems:
- Lower level: spatially-distributed data described as a field.
- Higher level: abstract geometric/topological structures uncovered in the data.

The higher-level abstract descriptions then serve as the basis for tasks such as inferring behaviors and designing controls. 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 from Section II-A.5, 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 order to bridge the gap between input field and abstract structural description, physics-based encapsulation relies on physical knowledge such as continuity and locality. The input data and underlying physical process must exhibit these properties in order for physics-based encapsulation to be applicable. For example, it is hard to extract structures in a very smooth image, or in a white-noise image.

In summary, physics-based encapsulation is suitable for a wide range of tasks expressible in terms of the discovery and manipulation of geometric and topological structures in spatially distributed data.

C. How is Physical Knowledge Encapsulated in Application Design?

The goal of physics-based modeling 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.

Again, in the trajectory bundling example, structures at different scales include sample points, trajectories, and bundles of trajectories. The structures in the force allocation of the air-jet system include the individual jets, the virtual jets as defined in Fig. 11, and the board-level virtual jet; the structures in sensing include pixels, strips, and groups of strip sensors that detect the sheet boundary.

2. Choose abstraction levels based on the relationships between these structures. 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 air-jet example, the edge-crossings are a substructure of sheet boundary.

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. In the air-jet example, the sensor fusion occurs progressively at the pixel, edge-crossing, and sheet boundary levels. Alternatively, the hierarchy could be specified top-down by noticing that bundles are comprised of curves which are comprised of points. In the air-jet example, the force allocation starts at the entire table, recursively decomposes it into the jet modules, and finally ends at the individual jets.

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. In the air-jet example, the edge-crossings can be grouped together if they fit with a hypothesized boundary of the sheet in the least-square sense.

D. What Modeling and Design Discipline Is Followed?

A physics-based design navigates through the 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 physics-based encapsulation at the beginning of this section):

• *Distributed data*: Physics-based design manipulates spatially distributed data, for example from a set of sensors or actuators. 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*: Physics-based design exploits the fact that physical interactions propagate from local to global, in order to build programs that reason from local to global. In particular, the *Ngraph* 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, physics-based design exploits 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 uses *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).

E. How Are Component Implementations Chosen and Manipulated?

The final step in physics-based software 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 a wave problem using a finite-difference method.
- *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 are defined by:

- *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. The computation of sensor fusion occurs in a similar fashion. Local pixel detections in a single sensor strip are combined into an edge-crossing first. The nearby edge-crossings are then combined by a line fitting.

V. Conclusion

We have presented the Spatial Aggregation framework of physics-based encapsulation for massively distributed embedded sensing and control applications. SAL encapsulates data and associated computation with physics-based "spatio-temporal objects". SAL programming constructs provide a vocabulary for expressing and utilizing the appropriate physical knowledge (distance metrics, locality constraints, similarity metrics, and so forth) to create and transform spatio-temporal objects. This physics-based encapsulation approach supports explanation of and meta-level reasoning about control decisions in terms of the compile-time physical knowledge and the run-time structural descriptions of the input data.

The case study of the air-jet system has demonstrated that the encapsulation of physics and task requirement as virtual sensors and actuators can drastically simplify the design of sensor fusion and actuation allocation algorithms. The encapsulation also improved the modularity of the embedded software by separating the logical level control design from physical layout and implementation of the sensors and actuators.

Acknowledgments

The work on the air-jet system reported here has been carried out in collaboration with a number of researchers at PARC, including Andy Berlin, Marshall Bern, David Biegelsen, Patrick Cheung, David Goldberg, Warren Jackson, Bryan Preas, and Jim Reich.

The authors thank Jim Reich, Jie Liu, Julia Liu, Jim Kurien, and Xenofon Koutsoukos for helpful discussions and comments on drafts of the paper.

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract numbers F33615-99-C-3611 (SEC Program), F30602-00-C-0139 (SensIT Program), F33615-01-C-1904 (NEST Program), and DABT63-95-C-0025 (MEMS Program), and in part by NSF grant CCR-9457802, ONR grant N00014-97-1-0599, and a Sloan Research Fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

[1] A. Huang, C-M. Ho, F. Jiang, and Y-C. Tai, "Mems transducers for aerodynamics — a paradym shift," in *AIAA 96-4017, 38th Aerospace Sciences Meeting & Exhibit*, Reno, NV, Jan. 10-13 2000.

[2] A. A. Berlin, D. K. Biegelsen, P. Cheung, M. P.J. Fromherz, D. Goldberg, W. B. Jackson, E. Panides, B. Preas, J. Reich, and L. E. Swartz, "Paper transport using modulated air jet arrays," in *IS&Ts NIP 15: 1999 International Conference on Digital Printing Technologies*, 1999, pp. 285–288.

[3] W. B. Jackson, M. P.J. Fromherz, D. K. Biegelsen, J. Reich, and D. Goldberg, "Constrained optimization based control of real time large-scale systems: Airjet object movement system," in *40th Int. Conf. on Decision and Control*, Orlando, Florida, Dec. 2001.

[4] Michael Tiller, *Introduction to Physical Modeling With Modelica*, Kluwer Academic Publishers, 2001.

[5] A. Mukherjee and R. Karmakar, *Modeling and Simulation of Engineering Systems Through Bondgraphs*, CRC Press LLC, Boca Raton, FL, 1999.
[6] K.M. Yip and F. Zhao, "Spatial aggregation: theory and applications," *Journal of Artificial Intelligence Research*, vol. 5, 1996.
[7] C. Bailey-Kellogg, F. Zhao, and K. Yip, "Spatial aggregation: language and applications," in *Proceedings of AAAI*, 1996.
[8] K.M. Yip, "Structural inferences from massive datasets," in *Proceedings of IJCAI*, 1997.
[9] F. Zhao, "Intelligent simulation in designing complex dynamical control systems," in *Artificial intelligence in industrial decision making, control, and automation*, Tzafestas and Verbruggen, Eds. 1995, Kluwer Academic Publishers.
[10] K.M. Yip, F. Zhao, and E. Sacks, "Imagistic reasoning," *ACM Computing Surveys*, vol. 27(3), 1995.
[11] X. Huang and F. Zhao, "Finding structures in weather maps," Tech. Rep. OSU-CISRC-3/98-TR11, Ohio State University Department of Computer and Information Science, 1998.
[12] F. Zhao, "Extracting and representing qualitative behaviors of complex systems in phase spaces," *Artificial Intelligence*, vol. 69(1-2), pp. 51–92, 1994.
[13] L. Joskowicz and E. Sacks, "Computational kinematics," *Artificial Intelligence*, vol. 51, pp. 381–416, 1991.
[14] C. Bailey-Kellogg and F. Zhao, "Influence-based model decomposition," *Artificial Intelligence*, vol. 130, no. 2, pp. 125–166, 2001.
[15] K.M. Yip, *KAM: A system for intelligently guiding numerical experimentation by computer*, MIT Press, 1991.
[16] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.
[17] J. Munkres, *Elements of Algebraic Topology*, Addison-Wesley, 1984.
[18] D. K. Biegelsen, P. Cheung, L. E. Swartz, W. B. Jackson, A. A. Berlin, and R. Lau, "High performance electrostatic air valves formed by thin-film lamination," *Microelectromechanical Systems (MEMS), American Society of Mechanical Engineers*, vol. 1, pp. 163–168, 1999.
[19] M. P.J. Fromherz and W. B. Jackson, "Force allocation in a large-scale, distributed active surface," *IEEE Trans. on Control Systems Technology (submitted)*, Dec. 2001.