Formal Foundations of Sensor Network Applications

Jacob Beal[1], Mirko Viroli[2]
[1]Raytheon BBN Technologies, USA, [2]Universita di Bologna, Italy

Abstract

One of the key features that distinguishes sensor networks from other networked applications is that their focus is generally not the sensors per se, but space-filling phenomena of the environment through which the sensors are deployed. Following the mathematical implications of this observation leads to a formal grounding of sensor network applications in a field calculus *that describes sensing, modeling, and interpretation of space-filling phenomena directly in terms of operations on mathematical fields. This points to more flexible, scalable, and resilient approaches to sensor network applications, as well as simpler approaches to developing decentralized applications that can provide robust services in difficult operating environments such as natural disasters, mass events, and critical cyber-physical systems.*

1 From Sensors to Fields

The advent of cheap computer networking has transformed sensing applications, allowing a dramatic increase in both the number of sensors deployed and the speed with which sensor information can be accessed. The models used for constructing networked sensing applications, however, are still largely encumbered with the assumptions of generic computer networking.

Under a generic networking model, the focus is on individual devices, each of which may be a source of unique data services or user requirements. In sensor networks, on the other hand, each individual device is more often simply one sample of a space-filling environmental phenomenon, such as the temperature of the air, the flood stage of a river system, or the distribution of an invasive species. The fact that these space-filling phenomena are sensed with particular sensors at particular locations is of relatively little concern: the value of the individual sensors is their ability to help model the phenomena of interest. Even the care taken to ensure good placement of sensors is indicative not of their importance, but of their unimportance, as the aim is generally to ensure that the placement of sensor provides the most representative view of the phenomenon.

We would likely be well-served, then, to engineer sensing applications using models that consider not the samples, but the continua that they represent. Formally, both space-filling phenomena and collections of sensor measurements can be formalized as mathematical *fields*, where a field is a function that maps each point in some region of space to a data value. Sensor applications may then be specified at an aggregate level, modular with respect to the details of how sensors are distributed and networked, in terms of the acquisition and manipulation of fields. For example, the collection of stream gauges making up the Iowa Flood Information System (IFIS) [6] may be thought of as sampling a field of sonar ranging data, which is then compared point-wise with another field of baseline measurements to produce the desired field: the estimated current flood stage at every location along the river systems that are being monitored.

Following the implications of this mathematical formalization has potentially profound impacts on the development of sensor applications. Formalizing data as fields leads to a formal field-based model of computation, the *field calculus* (Section 2), which in turn can form the foundation for aggregate programming models that

implicitly provide sensor network applications with flexibility, scalability, and resilience (Section 3). This in turn can enable simple design of decentralized applications for providing robust services in difficult operating environments such as natural disasters, mass events, and critical cyber-physical systems (Section 4), opening up possibilities for new high-impact applications in sensor-driven services and cyber-physical systems (Section 5).

2 Field Calculus Systematizes Computation with Fields

The observation that many applications can be thought of in terms of space-filling fields has been widely shared, not only for sensor networks, but in a wide array of other domains as well, including swarm robotics, biological modeling, and pervasive computing. This has led to the development of a large variety of approaches to programming at an aggregate level, including some tailored for sensor networks such as as Regiment [12] and TinyDB [9]: a thorough survey may be found in [2], and another focused on approaches specifically for sensor networks in [11]. Most of these approaches, however, have been rather ad hoc and specialized around particular assumptions about network structure and goals, making them generally difficult to apply pragmatically.

The *field calculus* [14] provides a more solid mathematical grounding derived from the common concept of "field" shared by many of these approaches and the commonalities in how such fields are manipulated. Its aim is to provide a "core calculus" that captures the essence of programming with space-time fields while remaining small enough for mathematical analysis to be tractable—just a λ-calculus does for functional programming [5], π-calculus for parallel computation [10], and FJ for object-oriented programming [7].

In particular, field calculus computation use an "everything is a field" paradigm: variables, computed values, sensed values, values feeding actuators, intermediate results of computations—all of these are fields. So in a sensor/actuator network we have the field of temperature detected on the ground, the field of users as detected by GPS sensors on smartphones, the field of directions as detected by accelerometers on smartphones, the field of identifiers for the Bluetooth beacons spread in a building, the field of Boolean values to activate lights in a smart-city, the field of vectors to guide unmanned vehicles or drones, etc. A computation thus starts from input fields, which are sensed environmental values or constants, and ends up with output fields representing information, either simply representing a model of the phenomena or being further fed to drive actions of the system on the basis of that model. Such a computation is carried on by progressively composing inputs using five basic constructs: "built-in" functions involving no communication or memory (e.g., addition, cosine, square root, local sensing or actuation), communication by sharing state with neighbors (operator nbr), memory via a state update construct (operator rep), branching by restriction of field domains (operator if), and calls to user-defined functions that abstract such combinations of constructs.

Critically, field calculus is both universal and has aggregate-local equivalence. Universality means that it can approximate any function on either a discrete network or a continuous region of space to an arbitrary level of precision [4]. At the same time, the particular choice of operations means that we can freely shift between aggregate and per-device models of computation, meaning that computations specified in terms of fields can be automatically transformed into equivalent programs to be run on the individual devices of a sensor network. Together, these two properties mean that field calculus supplies a general mathematical foundation for both implementation and analysis of any arbitrary sensor network application.

Finally, this mathematical framework has been implemented in a practically usable form by its instantiation in Protelis [13], a Java-hosted implementation of field calculus that allows simple integration of field calculus programs with the full range of Java libraries. As a simple example, the following code shows a user-defined function: `gossipMin` computes the minimum value of the input that has been sensed at any time at any location, and `boundedGossipMin` bounds this computation to the only that portion of space where `region`

holds true:

```
// Gossiping is achieved by repeated updates of variable v to the minimum observed either locally or across neighbours
def gossipMin(value) {
  rep(v <- value) {  min (value, minHoodPlusSelf(nbr(v)))  }
}
// Branching construct if restricts the domain of each branch to a subspace, with no communication between subspaces
def boundedGossipMin(value,region) {
  if(region) {  gossipMin(value) } else { null }
}
```

3 From Field Calculus to Practical Aggregate Programming

While field calculus is universal, it is also too low level for practical use in building complex distributed services. Mostly, this is due to the fact that any field computation that involves medium- to long-range propagation of information needs to suitably combine three different constructs (as shown in the `gossipMin` example): construct `nbr` to propagate information across a single step away, an built-in accumulation function (like `minHood`) to reconcile different values coming from different neighbors, and `rep` to chain information from step to step across a longer distance. Any non-trivial algorithm of collective behaviour, then, tends to require multiple stacked levels of such propagations, and can quickly become quite complex to build, read, debug, and maintain. This should, however, be seen as a flaw of field calculus itself: rather, it is the common characteristic of all core calculi, since their goal is not ease of programmability but the smallest set of ingredients necessary to support expressiveness in principle, as such terseness is vital for mathematical analysis of the properties of the calculus.

Instead, core calculi are generally rendered usable by using them to construct code libraries that raise the level of abstraction to a much more natural programming interface. For field calculus, we thus begin to raise the level of abstraction by identifying a collection of general "building block" operators for constructing resilient coordination applications, making the construction of increasingly complex services more practical. Each of these building blocks captures a family of frequently used strategies for achieving flexible and resilient decentralized behavior, hiding the complexity of using the low-level constructs of field calculus. Moreover, by using these building blocks exclusively, one can guarantee that the resulting system has some formally proven resilience properties [3].

Raising the level of abstraction yet further, on top of the building blocks one can stack additional layers of libraries (as shown in Figure 1), moving from general-purpose elements of self-organisation up to more specific mechanisms of collective adaptive behavior, and ultimately to sensor or sensor/actuator applications.

The critical layer of building blocks is formed by three new general operators G, C and T, along with field calculus' if and built-ins, which behave as follows:

- G is a "spreading" operation generalizing distance measurement, broadcast, and projection, which takes as input four fields: `source` (a Boolean indicator field), `init` (initial values for the output field), `metric` (a function providing maps associating a distance to each neighbor), and `accumulate` (a binary function over values. It may be thought of as executing two tasks: it computes a field of shortest-path distances from the `source` region according to the supplied function `metric`, then propagates values along the gradient of the distance field away from source, beginning with value `initial` and accumulating along the gradient with `accumulate`. As an example, if `metric` is physical distance, `initial` is 0, and `accumulate` is addition over floats, the G creates a field mapping each device to its shortest distance to a source [8, 1].

- C is complementary to G, it accumulates information down to the gradient of a supplied `potential` field. It takes as input four fields: `potential` (a numerical field), `accumulate` (a binary function

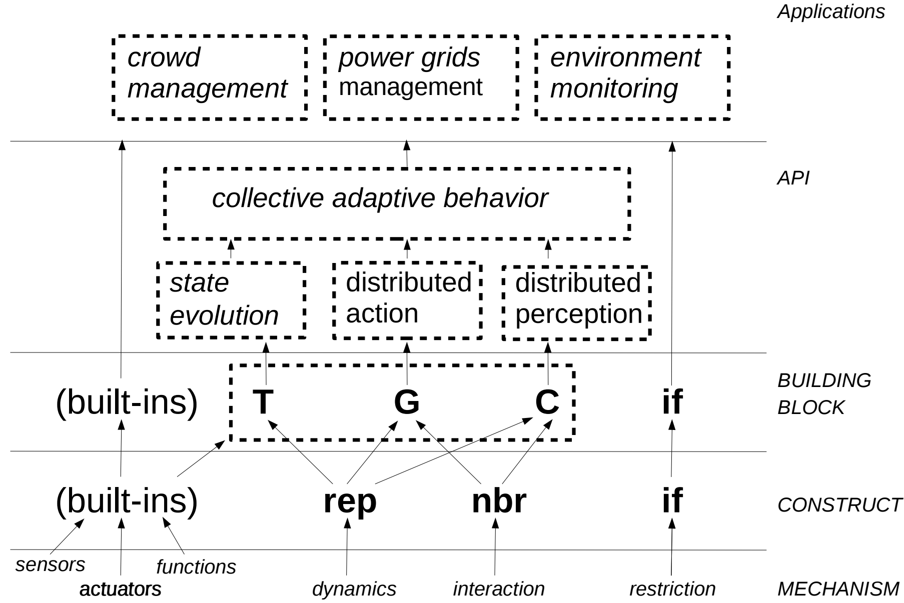


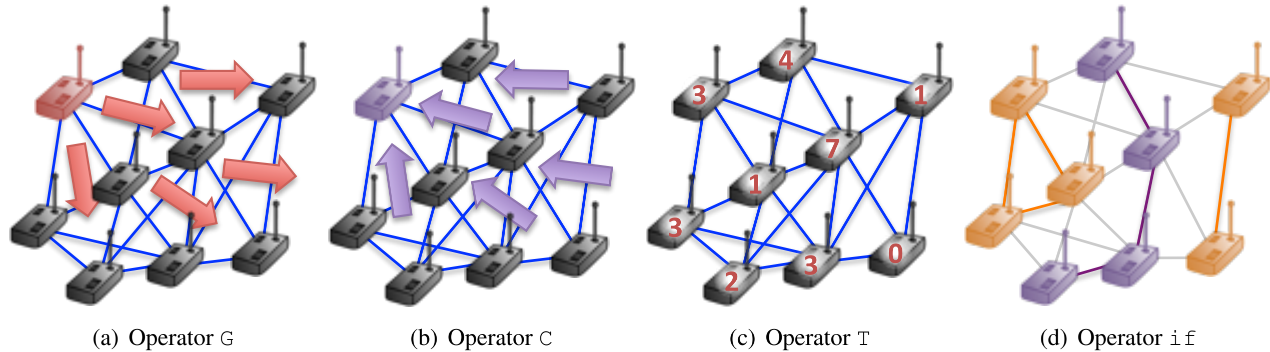
Figure 1: Layers implementing aggregate programming

over values), `local` (values to be accumulated), and `null` (an idempotent value for `accumulate`). Beginning with an idempotent `null`, at each device the `local` value is combined with "uphill" values on the `potential` field using a commutative and associative function `accumulate`. For instance, if `potential` is exactly a distance gradient computing with G in a given region R, `accumulate` is addition over floats, and `null` is 0, then C collects the sum of values of `local` in region R.

- T deals with time, whereas G and C deal with space—since time is one-dimensional, it carried out the functions of both spreading and collecting. It takes as input three fields: `initial` (initial values for the resulting field), `zero` (corresponding final values), and `decay` (a unary operator over values). Starting with `initial` at each node, that value gets decreased by function `decay` until eventually reaching `zero` value. It is hence a flexible count-down toward zero, where the rate of the count-down may change over time, and on top of which any local state evolution can be implemented. For instance, if `initial` is a pair of a value v and a timeout t, `zero` is a pair of an unknown value `null` and 0, and `decay` takes a pair to remove the elapsed time since previous computation from second component of the pair, and turning the first component to `null` if the first reached 0, then T implements a limited-time memory of v.

It can be shown [3] that when properly implemented, these building blocks, plus if and built-in operators, enjoy the following properties, which are then naturally inherited by APIs and applications built on top of them: **stabilisation:** if the input fields eventually stabilise to a fixed state, the same happens for the output field; **resilience:** if some messages get lost during system evolution, or some node temporarily fails, this will not affect the final result; **adaptability:** if input fields or network topology changes, the output field automatically adapts and changes its shape accordingly; and **scalability:** the performance of output field evolution is not affected by increase of the number of devices, provided that computation frequency and range maintain equivalent ratios.

On top of such building blocks, it is then possible to build libraries for addressing the various concerns of decentralized sensor network applications, including libraries for state evolution (e.g., timers, timed memory, integrators over time), distributed action (e.g., gradients, path forecasting, network partitions, broadcasts), distributed observation (e.g., summary, average, integral over regions), and more generally, collective adaptive behaviour (e.g., distributed situation recognition, leader election, collective choice, anticipative adaptation).



(a) Operator `G` (b) Operator `C` (c) Operator `T` (d) Operator `if`

Figure 2: “Building block” operators for distributed services: G spreads information from some sources outwards, progressively computing information en-route, C aggregates information in a given region by progressively transmitting it to a sink area, T aggregates over time by evolving a state variable until a fixpoint is reached, and if performs a distributed branch, partitioning the domain into non-interacting subspaces in which different computations carry on.

As an example, in [3] it is shown how a specification of only around 20 lines of code, using a library API based on such building blocks, defines an entire decentralized application for crowd sensing at mass events via opportunistic interactions between personal devices: operator G (via more user-friendly library functions) is used to sparsely partition the space into clusters of a characteristic size, operators C and T are used to model the crowd size and density within each cluster, and then G applied again to warn of dangerously crowded areas, suggest directions of dispersal, and provide navigation services for circumventing dense crowds.

4 Applications of Aggregate Programming

Shifting from foundations to applications, the areas in which field-based computational models are likely to have the most impact are those in which decentralized data processing is valuable. Decentralized data processing is much more complex than centralized, cloud-based processing, however, and so is only likely to be embraced when there is a strong need for it. In practice, the rapid expansion of civil wireless infrastructure means that in many cases, there is no need for decentralized data processing: building WiFi, fast data over phone cellular networks, and cheap satellite communication via systems like Iridium means that in many applications, sensors can simply exfiltrate data directly to cloud facilities and remove any need for considering the additional complexity of decentralized processing.

In a number of critical application areas, however, the demand for communication badly outstrips the expected available resources, and decentralized data processing makes sense. The three main cases in which this applies are:

- **Wireless infrastructure is inherently lacking:** In some scenarios, a strong wireless infrastructure simply cannot be assumed to be available, such as natural disasters (in which the infrastructure may be damaged) or underwater systems (in which high-speed communication simply is not available). There are also scenarios such as military operations where wireless infrastructure exists, but must not be relied upon.
- **Extremely high density of information:** In other scenarios, wireless infrastructure exists and may be accessible, but the density of sensors or their rate of information generation is too high to be supported. Examples include mass crowd events such as marathons and street festivals, where the number of attendees overwhelms the cell phone infrastructure, or emerging sensor-embedded materials, such as smart fabrics or adaptable wing surfaces, in which there may be a very large number of sensors taking data at high rates.

- **Tight sensor-actuator loop:** The third main case is scenarios where on average the network may be able to sustain the communication, but safety-critical actions need to be taken in a timely fashion on the basis of the sensed data. Examples include UAVs and urban traffic management. In this case, the system cannot tolerate even occasional network slowdowns or disconnection, and at least some processing must be decentralized for the sake of safety.

In all of these types of scenarios, field-based computation models can be valuable because they separate the concerns of managing a resilient networking and adaptation to changes in the set of sensors from the more global application-level concerns, making many aspects of resilience and scalability implicit and thereby greatly simplifying the engineering of decentralized applications.

Even when processing is centralized, however, field-based computation models may prove useful. The same parallelism that allows them to be executed in a decentralized manner can also allow a field-based computation to be implicitly parallelized for faster processing on a server or cloud system. Finally, its adaptability to changes in resolution may also allow for mixed models, in which a fast, coarse model is computed in a decentralized manner to ensure rapid and non-disrupted services, while a higher-precision model is computed with somewhat more delay in the cloud, and supplied as it becomes available.

5 The View Forward

At a fundamental level, sensor network applications are generally well-represented as computation on space-filling fields of sensor information. The details of network interactions used to implement them are only important insofar as they support or interfere with this computation, and thus those details should be separated as much as possible from the specification of the sensor network application.

Using a field-based model of computing can facilitate this separation. When combined with libraries of self-organizing building block algorithms, this can greatly simplify the implementation of decentralized network applications, thereby expanding the applicability of sensor network applications in large-scale and difficult domains such as natural disaster response, mass events, and control of cyber-physical systems. Software support for making use of these methods is still early, but the simple mathematical core of field calculus and the open-world Java integration supported by Protelis significantly lower the cost of adoption compared to prior aggregate programming methods. With this better understanding of the formal foundation of sensor network applications and their consequences, we may thus look forward toward a future in which sensor network applications are increasingly scalable, resilient, and tightly integrated with the everyday operations of society.

Acknowledgments

The views expressed in this manuscript have benefited from a number of valuable conversations with the participants of the Dagstuhl Seminar "Geosensor Networks: Bridging Algorithms and Applications," organized in December 2013 by Matt Duckham, Stefan Dulman, Jorg-Rudiger Sack, and Monika Sester.

This work has been partially supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873 (Viroli), by the Italian PRIN 2010/2011 project "CINA: Compositionality, Interaction, Negotiation, Autonomicity" (Viroli), and by the United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-10-C-0242 (Beal). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings contained in this article are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release; distribution is unlimited.

References

[1] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975. ACM, 2008.

[2] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

[3] J. Beal and M. Viroli. Building blocks for aggregate programming of self-organising applications. In *Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8-12, 2014*, pages 8–13, 2014.

[4] J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.

[5] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.

[6] I. Demir and W. F. Krajewski. Towards an integrated flood information system: centralized data access, analysis, and visualization. *Environmental Modelling & Software*, 50:77–84, 2013.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

[8] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.

[9] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[10] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[11] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19, 2011.

[12] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, pages 78–87, Aug. 2004.

[13] D. Pianini, J. Beal, and M. Viroli. Practical aggregate programming with PROTELIS. In *ACM Symposium on Applied Computing (SAC 2015)*, 2015. To appear.

[14] M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Sci.*, pages 114–128. Springer Berlin Heidelberg, 2013.