PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Review

Cite this article: Beal J, Viroli M. 2015 Space–time programming. *Phil. Trans. R. Soc. A* **373**: 20140220.
http://dx.doi.org/10.1098/rsta.2014.0220

Accepted: 17 April 2015

One contribution of 13 to a Theo Murphy meeting issue 'Heterotic computing: exploiting hybrid computational devices'.

Subject Areas:
artificial intelligence

Keywords:
spatial computing, field calculus, aggregate programming

Author for correspondence:
Jacob Beal
e-mail: jakebeal@bbn.com

Space–time programming

Jacob Beal[1] and Mirko Viroli[2]

[1]Raytheon BBN Technologies, Cambridge, MA, USA
[2]Alma Mater Studiorum, Università di Bologna, Bologna, Italy

Computation increasingly takes place not on an individual device, but distributed throughout a material or environment, whether it be a silicon surface, a network of wireless devices, a collection of biological cells or a programmable material. Emerging programming models embrace this reality and provide abstractions inspired by physics, such as computational fields, that allow such systems to be programmed holistically, rather than in terms of individual devices. This paper aims to provide a unified approach for the investigation and engineering of computations programmed with the aid of space–time abstractions, by bringing together a number of recent results, as well as to identify critical open problems.

1. Introduction

A major transition is ongoing in the way in which we interact with computing devices. Historically, the cost and complexity of computers had ensured that they were distributed only sparsely in space, except in certain special cases such as data centres. Thus, even though computers are of course physically implemented, it has been reasonable to use programming models that treat a computer as an abstract device, only incidentally connected to the physical world through inputs and outputs.

These assumptions, however, are breaking down in a number of ways. Rather than individual devices, we increasingly require aggregates of devices, co-located in space and time, to cooperate in accomplishing computing tasks. Major trends in this direction include the following.

— Decreasing cost and size of electronics are leading to a vast increase in the number of wireless networked computing devices, from phones and tablets to smart lighting, point-of-service terminals and more. Desired interactions between devices are often highly localized,

THE ROYAL SOCIETY PUBLISHING

covering such diverse uses as tap-to-pay, Bluetooth-enabled multiplayer games, augmented reality and home automation.
— The continuing expansion of graphics processing units and multi-core processors means that even an individual modern 'computer' is now a collection of many computing devices, and the performance of this collective depends critically on the relationship between information flow patterns and the relative spatial locations of the devices.
— Advances in materials science and biology are allowing simple computing devices to be fabricated directly within novel substrates, such as fabric or living cells. These typically comprise vast numbers of extremely simple and error-prone computing devices, each interacting with only a few nearby neighbours yet needing to act coherently as a collective.

In all cases, the individual device is rapidly shrinking in importance: what is needed is to be able to reliably engineer collective behaviours out of such aggregates.

With ordinary programming approaches, which focus on individual devices, a system designer is faced with the notoriously difficult and generally intractable 'local-to-global' problem of predicting a possibly emergent collective behaviour from the specifications for individual devices. A better approach is to take an *aggregate programming* perspective, in which one specifies collective behaviour by composing 'building block' algorithms, each of which has a known mapping from collective behaviour to a set of local interactions that produce that behaviour. While theoretically there is no difference in the formal expressive power of aggregate programming and more conventional programming, these collective abstractions can greatly increase the effective power by disentangling various aspects of distributed system design and allowing some aspects to be handled automatically and implicitly. This can be particularly effective for spatially embedded systems, where geometric constructions provide a ready source of useful building block algorithms.

A number of different research communities have recognized this challenge and have developed a wild menagerie of domain-specific models for so-called space–time computation, namely, computation expressed (more or less explicitly) in terms of properties of the physical time and space in which it occurs—a thorough survey may be found in [1]. Recently, however, there have been a number of unifying results, which have the potential to lead to a more general and broadly applicable theory of aggregate programming and to effective tools for practical use of aggregate programming in a variety of application domains.

The aim of this paper is to draw these results together to provide a unified model for the investigation and engineering of space–time programming. Section 2 begins by discussing the relationship between discrete computational devices and the continuous space–time in which they are embedded. Section 3 then lays out a roadmap for mastering space–time computing systems through development of aggregate programming capabilities. Following this roadmap, §4 presents field calculus as a mathematical foundation for aggregate programming, and §5 builds on this foundation towards effective, composable libraries for engineering resilient distributed systems. Finally, §6 summarizes these results and discusses key open problems.

2. Discrete devices versus continuous space–time

Most space–time approaches to computation draw on the same fundamental observation: a network of interacting computing devices may be viewed as a discrete approximation of the space through which they are distributed [1]. At the same time, a wide variety of applications can be naturally phrased in spatial terms, e.g. using sensors to sample an environment, modelling a natural phenomenon with spatial and temporal extent, or providing a service to users within a spatial region. For such applications, programming abstractions that take advantage of this observation about the relationship between networks and space can be highly valuable. On the one hand, they can disentangle the specification of the application from the details of its distributed implementation on a network, also helping in the creation of systems more insensitive

to environment perturbations of various sorts. On the other hand, they can ultimately provide engineers with a set of high-level constructs (language features, application programming interfaces (APIs), middleware services) to implement complex robust systems in an effective way.

To exploit this relationship, however, we must resolve the conceptual tension between discrete devices executing a computation in discrete steps, and the continuous space–time environment in which this computation takes place. Some approaches attempt to avoid this tension either by specifying computation using purely continuous representations such as partial differential equations (e.g. [2,3]), or by considering only quantized actions in quantized spaces (e.g. cellular automata such as [4] or [5], modular robotics such as [6] or [7], or parallel computing such as with StarLisp [8]).

For many systems, however, such evasions are not practical. Computations often require distinctly discrete operations such as conditional branching or iterative state update. Complementarily, it is rarely the case that it is practical to operate a crystalline arrangement of many devices in perfect synchrony—even parallel computing is becoming more asynchronous. In order to take advantage of geometric constructs in programming spatially distributed systems, it is thus necessary to have some theoretical framework for mapping between a continuous region of space and time and a set of discrete computational events taking place on computational devices within that region. This mapping must generally be bidirectional: the mapping from continuous to discrete is needed in order to generate local execution rules for implementing aggregate-level geometric constructs, while the mapping from discrete to continuous is used to understand how the results of a computation can be applied to its environmental context, and to raise the abstraction level up to be less dependent on actual physical location of devices and timing of their executions.

One such mapping is to consider the collection of discrete devices and events as a sample of a continuous environment—devices as samples of space or events as samples of space–time. This type of approach is most readily applicable when devices are physically much smaller than the gaps that separate them, such as in sensor networks (e.g. [9,10]), which are naturally viewed as sampling an environment, or in swarm robotics (e.g. [11]). A complementary mapping considers the space as a geometric construct formed by stitching together a discrete collection of space-filling 'cells'. This type of approach, notably used in frameworks such as MGS [12,13] and Gro [14], is more commonly used in biological modelling and in topological investigations.

Most approaches to space–time programming have been relatively ad hoc and informal in how they actually specify the relationship between continuous space and discrete computation. Recently, however, there has been an effort to improve the coherence and mathematical grounding of these computational abstractions. In this paper, we will discuss two complementary examples of such computational abstractions for programming discrete networks of devices embedded in continuous environments. The first, called the amorphous medium abstraction, aims to address issues of scale and perturbation sensitivity by viewing a network of computational devices as a discrete approximation of a computational continuum. The second, computational fields, views computation in terms of operations on field functions that map from a domain (continuous or discrete) of devices to a range of values that may be held by devices in the domain. When these two complementary concepts are used together, they pave the way towards a sound methodology for the development of complex distributed systems, as described in the following sections.

(a) The amorphous medium abstraction

The amorphous medium abstraction [15] is a computational abstraction intended to help address two common issues in distributed algorithms: scalability and sensitivity to network structure. This abstraction, illustrated in figure 1*a*, is based on the view of a network of locally communicating devices as a discrete approximation of the continuous space through which they are distributed. Now consider an infinite series of such networks in which the density of devices increases, while the time between communications and distance over which devices communicate decrease proportionally. The limit of this series is a continuous space filled with

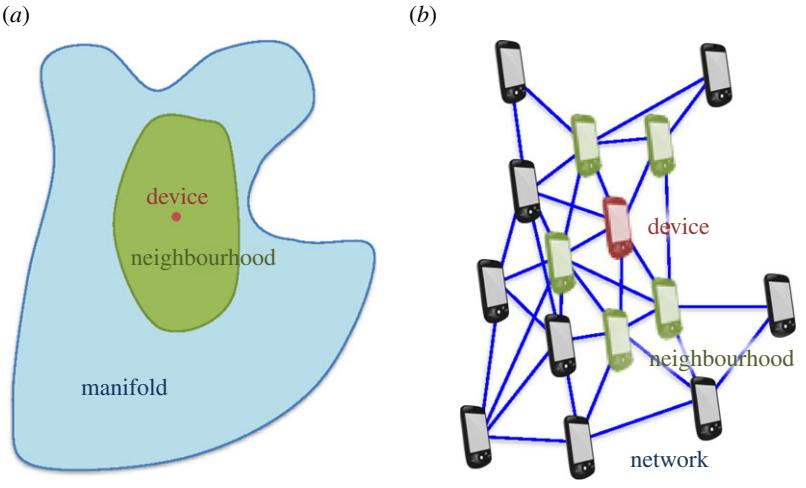

Figure 1. The amorphous medium abstraction (*a*) considers an infinite collection of computing devices, one at every point in a manifold, interacting by accessing the recent past state of devices within some neighbourhood of space. This can be approximated (*b*) by a network in which each device repeatedly broadcasts relevant states to its immediate network neighbours. (*a*) Continuous space and (*b*) discrete network. (Online version in colour.)

continuously computing devices and continuously propagating 'waves' of communication—in particular, the output of computation in each point can be viewed as a property of the space at that point, as explored in the next section. Because this continuum represents both extreme scale and a dissolution of discrete network structure, programs that can operate correctly both on this continuum and also in approximation are likely to have good resilience against both scale and the particulars of how devices are arranged in space.

More formally, the amorphous medium abstraction is defined as follows:

— the space is a Riemannian manifold, ensuring support for commonly used geometric constructs such as integrals, gradients, curvature and angles;[1]
— at every point in space, the abstraction assumes there is a universal computing device, forming an uncountably infinite set of computing devices (the limit as network density goes to infinity);[2]
— information propagates through the manifold continuously at no more than a bounded maximum velocity of c (representing hop-by-hop communication in a discrete network); and
— each device is associated with a local neighbourhood of devices no more than some distance r away (representing the locality of communication). A device then has access to the recent past state of any of its neighbours—in particular, for a neighbour at distance d, the device has access to state from d/c seconds in the past.

Obviously, such an uncountably infinite collection of devices cannot be physically implemented, but it can be approximated, as illustrated in figure 1*b*. In particular, computation specified for the abstraction of an amorphous medium can be approximated by a network in which each device repeatedly broadcasts relevant states to its immediate network neighbours. We may then quantify the continuous/discrete relationship by comparing the values of the approximation to the values that would be produced by computation on an amorphous medium [16].

[1]Though they are all suggested to be useful in practice [1], for some applications weaker properties may be sufficient and another class of manifolds could be substituted.

[2]Note that, yes, this does mean an amorphous medium is theoretically super-Turing; in practice, of course, only finitely approximable computations [16] appear to be realizable in our universe, and this subset is Turing-equivalent.

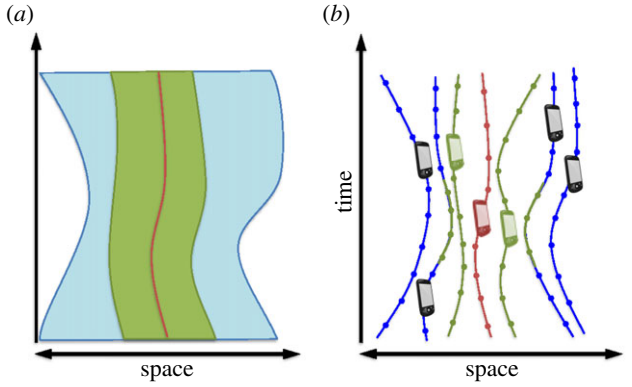

Figure 2. The amorphous medium abstraction illustrated in figure 1 may be extended to allow devices to move, in either continuous (*a*) or discrete (*b*) models, by considering each device to follow a trajectory in space–time, with its neighbourhood adjusting as it goes. (*a*) Continuous space–time and (*b*) network of mobile devices. (Online version in colour.)

Because it refers only to space, this definition of the amorphous medium applies only to non-mobile devices, in which it has been shown to be very useful. However, the model can be extended to support devices that may move in space, by considering each device as a sequence of events along a trajectory in space–time, the set of which comprises the manifold or network (illustrated in figure 2). In particular, it is necessary that devices do not intersect one another, either physically or in the continuous abstraction. In the physical world, the problem is the obvious physical reality of collision; in the continuous abstraction, it can require the manifold to hold two contradictory values at a single location. Additionally, in physics the movement of particles within space-filling materials depends on the state of matter that they are in: in a gas, there is enough space that particles can flow past one another; in a liquid, the presence of viscous forces is modelled by the vorticity functions, whose effect can be visualized as causing the particles to rotate past one another; while in a solid, most particles cannot move at all except through the propagation of vacancies through the material. These physical models may provide useful clues for how to approach mobile devices, as suggested in [17]—an open issue to be formally addressed in future works.

Regardless of these outstanding theoretical questions, the amorphous medium abstraction is a generalization over many other approaches to space–time programming [1], which are based on a continuous/discrete relationship that is at least somewhat compatible, if often less formally defined. The amorphous medium abstraction, however, only provides a way of relating continuous and discrete aggregate programs, but does not suggest how such programs might be specified. For that, we must turn to other complementary abstractions such as computational fields.

(b) Computational fields

Computational fields are another widely used space–time programming concept [1], recently formalized in the study of the semantics of Proto [18] and the computational field calculus [19]. The basic notion of a computational field is another example in which computer science looks towards physics to draw inspiration: a field in physics is a function that maps each point in some domain of space–time to a value, such as an electromagnetic potential or a force vector. Computational fields are the same, but may range over arbitrary computational objects. Thus, in addition to scalar fields, such as temperature, and vector fields, such as wind velocity, computational fields might also hold more complex data like sets, such as an inventory of items to be stocked on a store shelf, or records, such as XML descriptors of those inventory items. Of course, not all kinds of computational fields can be manipulated in the same way; for instance,

gradients are not well defined for non-numerical values. This is actually quite a standard situation in computer languages: operators should not be applied to all values, and this is readily enforced by a system of type constraints that restricts operators to be applied only to suitable input values.

The concept of a computational field is well defined for discrete networks as well. Discrete fields have a long history of usage in physics, e.g. in lattice gas models [20,21]. The particulars of an implementation may need to be adjusted, e.g. to cope with the differences between crystalline and amorphous networks, but the approach remains otherwise the same. Importantly, this means that computational fields may even be used for programming networks and applications that do not map well onto continuous manifold models, as developed in [22,23].

Given the basic concept of a computational field, aggregate computations can be constructed using operations on fields, taking fields as inputs and producing fields as outputs. Considered as an element of a program, a field is the distributed equivalent of a value or variable on a single device. A consistent programming model for this environment then provides a set of operations on fields that, when mapped to equivalent operations on individual devices, produces the same value as when the fields are manipulated directly as mathematical objects. From there, we may map a field computation to local execution by placing the value at each point in each field to its corresponding device in the network, and executing the equivalent local operations.

Computational fields are also a natural complement for the amorphous medium abstraction. Since the domain of a field may be either discrete or continuous, fields can be readily mapped from the abstract continuous model to realization on a discrete network. Thus for any network that approximates an amorphous medium, we may consider the approximation of a continuous field on the manifold by a discrete field on the network. Thus taken together, these two complementary concepts of amorphous medium and computational fields form the basis for the approach to space–time programming that is the focus of the remainder of this paper.

3. Aggregate programming

Programming robust spatially distributed systems is particularly challenging because of the several interacting aspects of distributed systems that must be simultaneously addressed:

— efficient and reliable communication of information between individual devices;
— coordination across many devices to provide robust services in the face of ongoing failures and changes in the network; and
— composition of subsystems and modules to determine which types of coordination are appropriate between which sets of devices in various regions of space and time.

Conventional programming methods focus on individual devices and their interactions, meaning that all of these aspects must be addressed simultaneously in the same program. A programmer developing a distributed application is thus challenged to solve a particularly complex version of the notoriously difficult local-to-global problem, of determining a set of per-device actions and interactions that will produce good behaviour for all of these aspects simultaneously. Since none of the aspects is isolated from the others, this often results in different concerns of coordination becoming threaded together within a single body of code. Space–time computations developed in this way thus tend to suffer from some combination of being badly performing, confusing to use, difficult to maintain and/or not readily decomposable for re-use of modules.

Aggregate programming approaches generally address this problem by providing abstractions that allow each aspect to be addressed separately, namely more easily supporting an implicit management of some non-functional aspects related to robustness.

— Device-to-device communication is typically made entirely implicit, either choosing a domain-optimized set of assumptions for managing trade-offs between efficiency and robustness or else providing higher level abstractions for controlling those trade-offs.

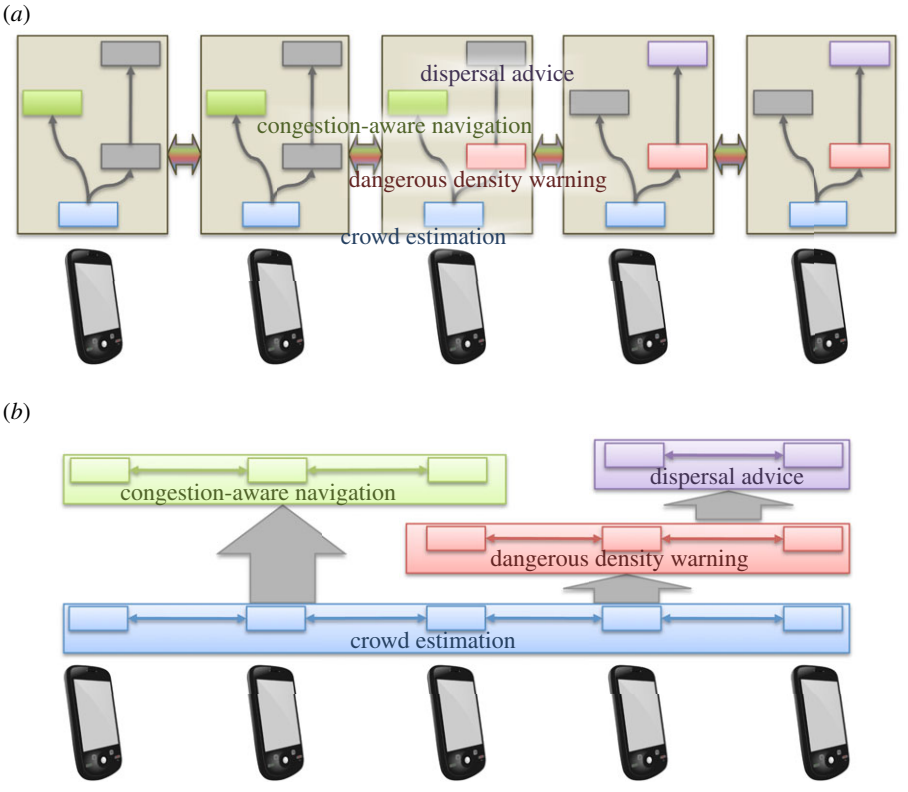

Figure 3. Diagrammatic representation of device-centric programming of distributed algorithms (*a*) versus aggregate programming (*b*). Device-centric programming designs a complex system in terms of the behaviour of a single device, while with aggregate programming, algorithmic building blocks can be scoped and composed directly for the aggregate. (Online version in colour.)

— Distributed coordination methods are encapsulated as aggregate-level operations (e.g. measuring distance from a region, spreading a value by gossip, sampling a collection of sensors at a certain resolution in space and time). In some cases, the set of available operations is fixed, while in others it can be extended using some sub-language for creating and encapsulating coordination methods.
— The overall system is specified by composing aggregate-level operations in various ways. This specification is then transformed into a complete distributed implementation by mapping each aggregate-level operation to its corresponding encapsulated coordination method, and these in turn use the provided implicit device-to-device communication model.

(a) Example: distributed crowd management

Figure 3 illustrates the difference between conventional and aggregate programming with an example of a crowd-management application, based on [23,24]. Such an application might be distributed to the cell phones of people attending a very large public event, such as a marathon or a major city festival [25]. In such environments, the communications infrastructure is often overwhelmed, while at the same time the movements of large crowds in relation to unanticipated obstacles and events can create dangerous situations that frequently result in injuries and deaths [26]. Since crowd density and motion are spatial phenomena, this is a good example of where spatially local communication between cell phones might be used to build an effective safety application for detecting high-density areas and helping to avert danger.

Figure 3 shows an example in which such an application is architected in terms of four interacting services: the core service is a distributed algorithm that estimates the crowd density function over space through local communication between cell phones. Based on this estimate, people near dangerously large and dense areas of crowd can be alerted, and the subset inside those areas given advice on how to safely move to alleviate the danger, while at the same time other parts of the crowd can be given density-aware navigation that helps people move around within the event space while avoiding highly congested regions.

Implemented with a device-centric programming approach, it is necessary to create a composite crowd-management application that implements a single-device fragment of all four services together. The information to be communicated between services must be collected and packaged for exchange between devices. Devices then decide which fragments of the composite application to run by examining both their local information and the information packages received from other devices.

With an aggregate programming approach, on the other hand, each service is designed and encapsulated separately. The crowd estimation service outputs a distributed model. Each of the other services then uses this model both as an input and to determine the subspace on which that service should run—i.e. the subset of devices that should execute the coordination methods that implement the service. This functional composition of services forms the complete application, whose communication details can be determined automatically by composing the communication requirements of the individual services.

The final set of actions and communications may be the same in both cases. The aggregate programming architecture is much cleaner and more modular than with device-centric programming, however, because the implementation of the coordination services has been separated from how they are composed and how their communication is implemented.

(b) Approaches to aggregate programming

The survey of methods for programming spatial aggregates in [1] identified a large number of distinct approaches. These various approaches consider a wide variety of different domains, from sensor networks and modular robotics to high-performance computing, from synthetic biology to pervasive embedded systems and more. Despite this diversity of methods and domains, however, all share the unifying challenge of space–time constraints on communication, and define computation over a distributed set of data elements (hence, implicitly or explicitly adhering to the notion of a computational field). This leads to the emergence of commonalities between the various approaches.[3] In particular, Beal *et al*. [1] found that aggregate programming methods for space–time computation tend to fall into four categories:

— *Device abstraction languages* do not actually address the local-to-global problem, but instead aim to allow a programmer to focus more clearly on grappling with it. These languages do this by by providing abstractions that implicitly handle other system details such as local communication protocols. Prototypical examples include NetLogo [5], Hood [27], TOTA [28], Gro [14], MPI [29] and SAPERE [30].
— *Pattern languages* allow aggregate programming of certain classes of geometric or topological constructions. Prototypical examples include Growing Point Language [31], Origami Shape Language [32] and the self-stabilizing pattern language in [33].
— *Information movement languages* gather information from certain regions of space–time, process it, and deliver it to other regions. The domain of sensor networks has particularly many of such languages, such as TinyDB [9] and Regiment [10].

[3]Some aggregate programming approaches actually neglect space–time constraints, as they rely on the non-situated parallelism typical of cloud-based architectures like Hadoop (http://hadoop.apache.org). As this paper focuses on spatially structured networks, however, this view is not considered further herein.

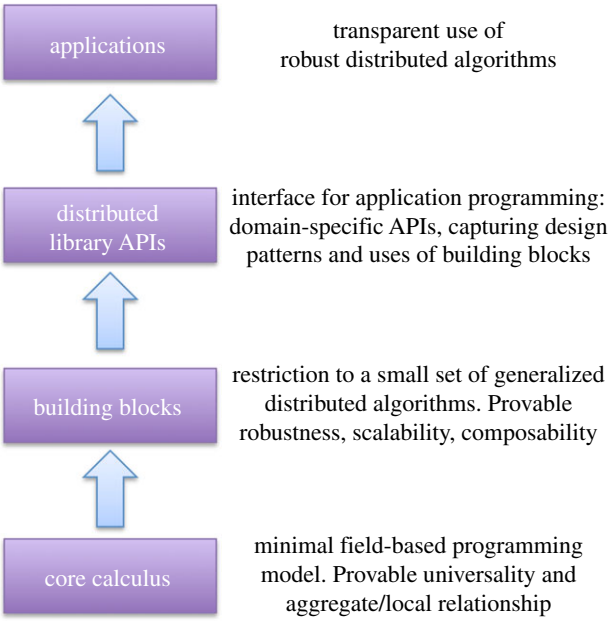

Figure 4. Layered roadmap for development of spatially distributed systems via aggregate programming: a core calculus with provable aggregate/local relations can be used to express a collection of general and composable 'building block' algorithms with desirable collective properties. Patterns of building-block use can then be captured in domain-specific libraries readily usable for distributed system construction. (Online version in colour.)

— *General purpose spatial languages* aim to provide domain-general methods for aggregate programming of space–time computations. These are the most recent to emerge and there are very few as yet, the most mature being Proto [34], Protelis [35], MGS [12] and field calculus [19], the last of which will be discussed in detail in the next section.

For the development of effective aggregate programming for real-world spatially distributed applications, it is necessary to focus on general purpose spatial languages. While pattern languages and information movement languages provide a number of elegant examples of aggregate programming, they are also generally sharply limited in the types of coordination methods (and thus computations) that they support. Real-world applications will typically be much more complex, often requiring many different types of coordination to be combined, as well as the ability to integrate smoothly with non-spatial aspects of the application, like user interaction, sensor processing, actuator control and cloud services. To address such challenges, we instead need general purpose spatial languages, which provide an open world that can be extended with new coordination methods and connections to external services, at the cost of providing less guarantees when these capabilities are used.

Figure 4 shows a layered approach to development of spatially distributed systems via aggregate programming with general purpose spatial languages. The foundation of this approach is a minimal core calculus, expressive enough to be a universal computing model but terse enough to enable a provable mapping from aggregate specifications to equivalent local implementations. One such model is the field calculus [19] presented in the next section. The next layer uses this minimal calculus to express a collection of higher level 'building block' algorithms, each being a simple and generalized basis element of an 'algebra' of programs with desirable resilience properties. Section 5 discusses the development of such a collection, and how patterns for using and composing building blocks can be captured in higher level library APIs, enabling simple and transparent construction of robust distributed systems.

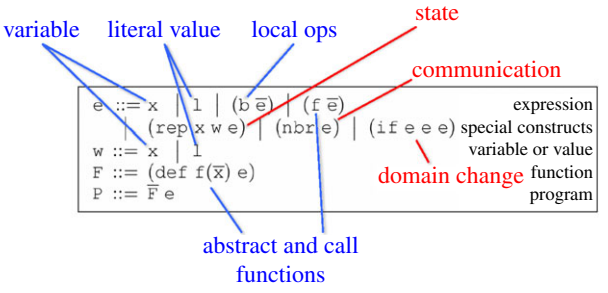

Figure 5. Field calculus is a minimal syntax and semantics for construction of aggregate programs with a coherent mapping from aggregate syntax to local syntax. (Online version in colour.)

4. Foundation: field calculus

Field calculus [19] provides a theoretical foundation for aggregate programming, grounding on the idea of computing by functional manipulation of computational fields. Its aim is to provide a universal model that is terse enough for mathematical proofs of general properties about aggregate programming and the aggregate/local relationship, just as λ-calculus [36] provides for functional programming, π-calculus for parallel programming [37] or Featherweight Java [38] for object-oriented programming.

(a) Syntax and semantics of field calculus

Field calculus [19] is a minimal expressive model for computation with fields, developed by refining the set of key operations in the wide range of domain-specific languages for space–time computation surveyed in [1] to find a small covering set. From this survey, it was decided to base field calculus on Proto [34], which is one of the few current general purpose space–time programming languages and which has quite general capabilities. Proto, however, has far too complicated a syntax and semantics (as developed in [39]) and is not practical for mathematical analysis. Field calculus was thus developed by finding a minimal set of operations that, with appropriate syntactic sugar, could cover all of Proto as well as certain capabilities not supported by Proto but supported by other languages.

As might be expected from its name, the value of any field calculus expression is a computational field ϕ, mapping from a potentially time-varying domain of devices to a value for each device at each point in time. From this very consideration, one derives that such an aggregate-level interpretation, which we shall follow in the remainder of this paper, is an improvement and extension over the classical pointwise interpretation of distributed system programming, in which the focus is still on the single device (in a given position of space) computing a single value (at a given time).

Fields are computed and manipulated according to the functional (Lisp-like) syntax shown in figure 5 (overbar notation denotes sequences, e.g. $\bar{\mathtt{e}}$ stands for $\mathtt{e}_1 \ldots \mathtt{e}_n$ with $n \geq 0$). Fields may either be literal data values (e.g. `5`, `'username'`) representing constant-valued fields over their domain, or else expressions that combine fields using five types of operation. The first two of these are generic, appearing in any functional programming language, while the other three (illustrated in figure 6) are special constructs for support of aggregate programming:

— *Built-in operators* are a collection of primitive operations (symbol `b`) taking in zero or more input fields and functionally computing an output field. Most specifically, the value of the resulting field at a device is the result of applying the built-in operator to the inputs at the same device. For example, multiplication of two numerical fields ϕ_1 and ϕ_2 yields a field ϕ mapping each device in space–time to $v_1 * v_2$, where v_1 and v_2 are the values to which ϕ_1

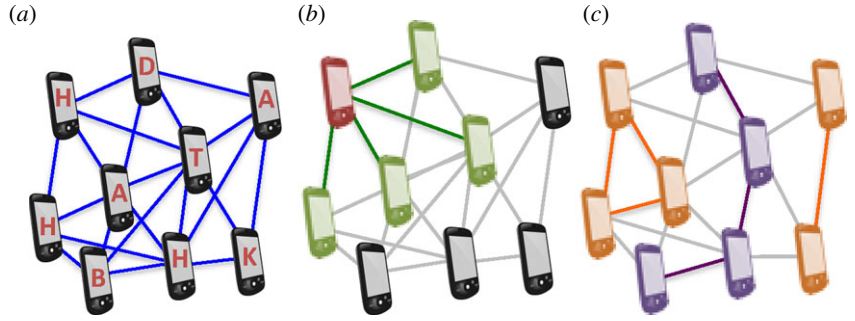

Figure 6. Special constructs of field calculus: (*a*) `(rep v i e)` creates a persistent state variable `v`, initialized to `i` and updated with expression `e` each round; illustration shows a state variable being used to remember a single character. (*b*) `(nbr e)` gives each device a collection of the values of `e` held by its neighbours; illustration shows the neighbourhood (green) of one of the devices (red). (*c*) `(if e t f)` splits the network/space into two subspaces, evaluating `t` where `e` is *true* (orange) and `f` where it is *false* (purple); grey links are ignored because they cross between the two subspaces. (Online version in colour.)

and ϕ_2 map that same device. Similar per-device definitions hold for other mathematical (e.g. addition and sine over numbers) and logical (e.g. conjunction and negation over Booleans) operators, or more generally for any operator that can be effectively computed with only local information (e.g. concatenation over strings, union over sets, minimum over the range of functions on finite domain). Additionally, built-in operators can be context-dependent, meaning that their result can be affected by the current state of the environment, or conversely can affect the environment. Examples include sensors (yielding a field of values, e.g. temperature), actuators (taking a field which feeds some actuator, e.g. to signal alerts on cell phones), and environment 'observers' (e.g. detecting close neighbours, as a field of sets of device ids).

— *Function definition and function calls* follow the standard pattern of functional programming: a declaration `(def f (`$x_1 \cdots x_n$`) e)` defines a function named `f`, which may then be called using `(f` $e_1 \cdots e_n$`)` to produce a value equivalent to evaluating body `e` with each variable replaced by its corresponding input (i.e. x_1 replaced by e_1, etc.).

— *Persistent state variables* are defined by the 'repeat' construct `(rep x w e)`, which is used to create fields that dynamically evolve over time. The `rep` construct initially sets variable `x` to the field value ϕ given by constant or variable `w`. At every 'distributed' computation step, this field value recomputes by evaluating `e` using the prior value of `x`. For example, expression `(rep x 0 (+ x 1))` denotes a field counting how many computations have happened at each device—it maps each device to 0 in the first round when that device is included in the domain, and it is incremented with each further computation where the device continues to remain in the domain.

— *Values from neighbours* are obtained with the construct `(nbr e)`, which produces a field of (finite-domain) functions as its output: to each device d it associates a map (or function) from d's neighbours to the value that each neighbour holds for `e` (which implicitly implies communication with those neighbours). Hence, the output of `(nbr e)` can be viewed as a field of fields, since the value of the field at each device d is itself a field. Note that the semantics of the field calculus do not allow nesting of `nbr` constructs to create fields of fields of fields (and the like). These collections of neighbour values can then be manipulated and summarized by built-in operators, such as `min-hood`, which creates a field of the minimum values in each device's neighbourhood.

— *Domain restriction* is a reduction in the set of devices over which an expression is evaluated. The construct `(if` e_0 e_1 e_2`)` takes a Boolean-valued field for e_0 (namely, a

field mapping devices to Boolean values) and computes e_1 only using a domain restricted to those devices where e_0 is *true*, and e_2 only using those where it is *false*.

Any field calculus program is then a sequence of function definitions, followed by a main expression to be evaluated. For example, consider a distributed sensor `temperature`, realized in the form of a numeric temperature sensor available in each device, hence producing a discrete scalar field that approximates the continuous physical scalar temperature field. A simple gossip computation that tracks the lowest temperature ever sensed by the aggregate may be defined as:

```
;; gossip-min computes a constant-valued field holding the minimum ever observed for ''value''
(def gossip-min (value)
  (rep lowest
    ;; The minimum starts as infinity ...
    infinity
    ;; ... and it keeps decreasing using the minimum value across neighbours.
    (min value (min-hood (nbr lowest)))))

;; main expression: obtain the minimum value of temperature ever observed
(gossip-min (temperature))
```

As a theoretical construct, the semantics of field calculus is defined in terms of a discrete network of devices evaluating a program across a discrete sequence of time points (full formal details are given in [19]). This is implemented by giving every device a copy of the expression, then performing periodic unsynchronized evaluations on each device. The `nbr` expressions are supported by having each device track the values it computes for each sub-expression during evaluation[4] and then broadcasting this information to its neighbours between evaluation rounds.

(b) Properties of field-based aggregate computations

Following the 'core calculus' approach that is traditional of the programming language community, the field calculus aims for a suitable trade-off between expressiveness and simplicity. Though a wide set of aggregate computations can be expressed, it also plays the role of a mathematical tool one can fruitfully exploit to: (i) define general properties of interest and prove their validity [40] (typically, by induction on the syntactic structure of an expression), (ii) isolate fragments with specific behavioural properties [41], (iii) design new mechanisms by extension [42], and (iv) pave the way towards sound implementation of tool support [35]. We here recap some of the basic properties that have been investigated so far:

— *Global–local coherence*. Coherence of the aggregate field computations in a distributed execution is maintained for field calculus by tracking which `nbr` operations align with one another, meaning that they are part of the same aggregate operation and should share information via the field output by `nbr`. Determining alignment has a number of subtleties stemming from the fact that execution may proceed differently on different devices, and that any given function may be called multiple times. As a consequence, two neighbouring devices may each evaluate a single `nbr` statement in the code many times, with some of these evaluations aligning and sharing information, while others do not align and remain isolated. Alignment is thus the local implementation of the aggregate notion of a field's domain, and the key to ensuring a provable relationship between

[4]In fact, it is sufficient to track only the values supplied to `nbr` operations and the context of those operations.

aggregate specification and local implementation. The minimal syntax of field calculus has allowed its semantics, including alignment, to be proven correct and consistent [19].

— *Self-stabilization*. What kind of behavioural properties can we analyse given an arbitrary field calculus program? This is quite a difficult problem when put in general terms, especially since field calculus can embed any computable function as a built-in operator, and hence suffers from the same undecidability properties of Turing machines. Isolating fragments enjoying certain specific properties of interest is however possible, with the potential of conceiving static analysis algorithms (and then, tools) supporting solid system engineering on top. This is what has been done in [41], where a fragment of the language in which constructs `rep`, `nbr` and built-in operator `min-hood` are used in a particular *monotonic* combination, forming the so-called 'stabilizing spreading expressions'. It is proved that in this fragment any program produces self-stabilizing fields, namely, fields that spontaneously adapt to transitory changes in the environment (device faults, changes to topology, changes to the information perceived from input sensors) until reaching a final state that will not change as time passes. Moreover, under rather weak fairness conditions on asynchronous device firing, time complexity of stabilization can be shown to be $O(d \cdot p)$, where d is the diameter of the network and p is the depth of the program expression.[5]

— *Universality*. The field calculus as proposed in [19] is given semantics on a discrete network of asynchronously firing devices. However, by associating to each computation round of a device in one such network a space–time event in a manifold, and considering the limit as device density and round rate increase towards infinity, it is possible to give a continuous semantics of the field calculus, as discussed in §2. In particular, in [40] it is proven mathematically that the field calculus is 'space–time universal': in other words, every causal and finitely approximable function over a Riemannian space can be approximated to an arbitrary degree of precision by some field calculus program, given a dense enough and fast enough network of devices.

These, then, are the key contributions of field calculus: any coordination method with a coherent aggregate-level interpretation is guaranteed to be possibly expressed in field calculus. Such a method can then be abstracted into a new aggregate-level operation using `def`, which can then be composed with any other aggregate operation using the rules of mathematical functions over fields, and can have its space–time extent modulated and controlled by `if`, all while guaranteeing that the relationship between global specification and local implementation will always be maintained.

Note also that although field calculus was developed for aggregate programming of space–time computations, there is nothing preventing it from being applied to more general network computations as well. Anything for which neighbourhood relationships can be defined is potentially programmable via field calculus, and it generally makes sense for any network in which the set of connections is sparse. Thus, for example, field calculus would not be good for the Internet in general, in which theoretically anything connects to anything, but should work well for the network of Internet backbone links carrying long-distance traffic, or for overlay networks built on top of general routing.

5. From building blocks to libraries

Core calculi are rarely used for actual programming: the same terseness that gives them value in proving mathematical properties also ensures that they are too low-level for system construction use, and field calculus is no exception. In addition, because it is universal it can express fragile and wasteful systems just as readily as resilient and efficient systems.

[5]Under stronger synchrony conditions, it can be shown that an exact upper-bound to stabilization time can be established once network shape, initial conditions and program structure are known (such time being still proportional to d).

It is thus necessary to raise the level of abstraction, building libraries with accessible APIs upon the foundation of field calculus. These can be decomposed further into two stages of library building. First, collect a small set of compatible 'building block' algorithms with as little overlap in functionality as possible, implemented in field calculus, that can be proved to be resilient and efficient and to maintain these properties when composed. Second, collect a variety of design patterns and typical uses of sets of building blocks to form libraries for application programming—likely a number of libraries customized for various domains.

More technically, at the 'building block' level, the aim is to identify an algebra of aggregate coordination operators, such that each operator has certain properties of resilience and scalability, and such that these properties are also preserved for any composition of operators. In effect, this is just like the field calculus guarantee of aggregate/local relations for any composition of field operators, except at a higher level of abstraction and aimed instead at ensuring certain properties of aggregate behaviour.

A significant first step towards establishing such an algebra is the taxonomy of self-organization methods identified in [43], though this first collection of operators did not investigate compositional properties. Similarly, the work in [41], as already mentioned, identifies 'usage patterns' of field calculus constructs that can result in self-stabilization.

Most recently, a richer collection of building blocks has been proposed in [24], aiming to be a first attempt at covering a taxonomy of space and time operation classes. This algebra uses five coordination operators and is self-stabilizing for all feed-forward combinations of operators. The operators are also implemented in field calculus, meaning that they inherit its guarantees of aggregate/local relations and can be extended, composed and modulated by the rules of field calculus.

One of the operators is restriction with `if` into two subspaces, as inherited directly from field calculus. The other four, illustrated in figure 7, are the following.

— `(G source initial metric accumulate)`. The `G` operator implements information spreading outward from the region indicated from Boolean-valued field `source`, along minimal paths according to distance function `metric`. The values spread start at `initial` and change as they go according to function `accumulate`. This operator can be used, among other things, to implement distance measures, broadcasts and forecasts.
— `(C potential accumulate local null)`. The `C` operator is an inverse of `G`, accumulating value `local` down the gradient of a potential field `potential`, starting with `null` and merging values according to function `accumulate`.
— `(T initial decay)`. The `T` operator tracks values over time rather than space, starting with value `initial` and decreasing towards zero over time according to function `decay`.
— `(S grain metric)`. The `S` operator implements sparse symmetry breaking, selecting a set of 'leader' devices such that no device is more than `grain` distance from a leader and no two leaders are within $\frac{1}{2}$ `grain` of one another, measuring distance by function `metric`.

Importantly, the proof of self-stabilization for these operators is based on showing that any system of operators with a certain self-stabilization property will be self-stabilizing under feed-forward composition, then showing that each of the five operators has this property. This is significant because it means that additional operators can be added to the system if needed without adding great complexity to the proof, if those operators can be shown to satisfy the same general property.

These five operators are, of course, quite abstract and general, meaning that although they are good for mathematical analysis they do not form a very user-friendly API for applications programming. More user-friendly libraries can be readily constructed, however, by defining field calculus functions that raise the abstraction level yet again, capturing typical design patterns and uses of the operators or combinations thereof. Being so constructed, such libraries also inherit all of the desirable properties of the building block operators.

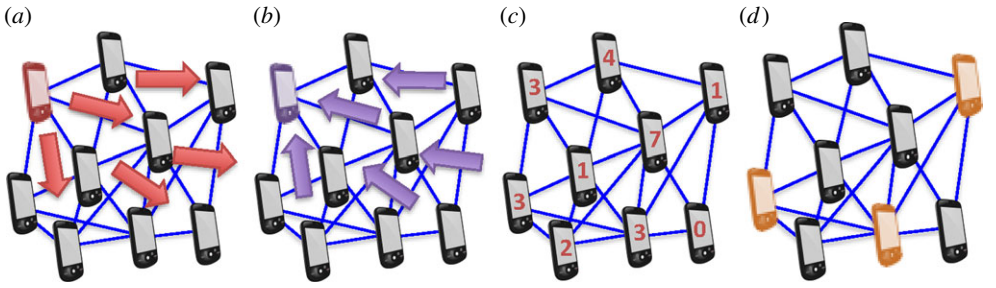

Figure 7. The algebra of 'building block' operators proposed in [24] comprises five operators: `if` (figure 6*c*), an information-spreading operator `G` (*a*), an information aggregation operator `C` (*b*), aggregation over time with operator `T` (*c*) and sparse symmetry breaking with operator `S` (*d*). Careful selection and implementation of these operators produces a broadly applicable algebra of self-stabilizing distributed systems. (Online version in colour.)

To illustrate this point, consider a few examples of implementing more user-friendly library functions out of building block operators, adapted from [24]. First is use of `G` to compute distance from a source, using the distance to neighbours `nbr-range` as a metric:

```
(def distance-to (source)
  (G source 0 nbr-range (fun (v) (+ v (nbr-range)))))
```

Similarly, combining `G` and `C` can accumulate values in a region to a designated sink device, then broadcast this summary value to every device in the region of space:

```
(def summarize (sink accumulate local null)
  (G sink
   (C (distance-to sink) accumulate local null)
   nbr-range identity))
```

And combining `if` and T can produce a limited-time memory that tracks whether a certain event has recently occurred at each point in space:

```
(def recent-event (event timeout)
  (if event
   true
   (> (T timeout (fun (t) (- t (dt)))) 0)))
```

Each of these examples captures a coordination pattern with a clear aggregate interpretation and a relatively simple implementation using field calculus and the five building block operators.

By collecting a large number of such library functions, it should be possible to create good APIs (general or domain-specific) for rapid development of reliable spatially distributed applications. For example, consider the crowd-steering application discussed in §3: figure 8 shows a simulation snapshot for an implementation of that application, taken from [24]. The simulation is of a scenario with 6500 people at an event in Boston's Columbus Park, 10% of whom have devices

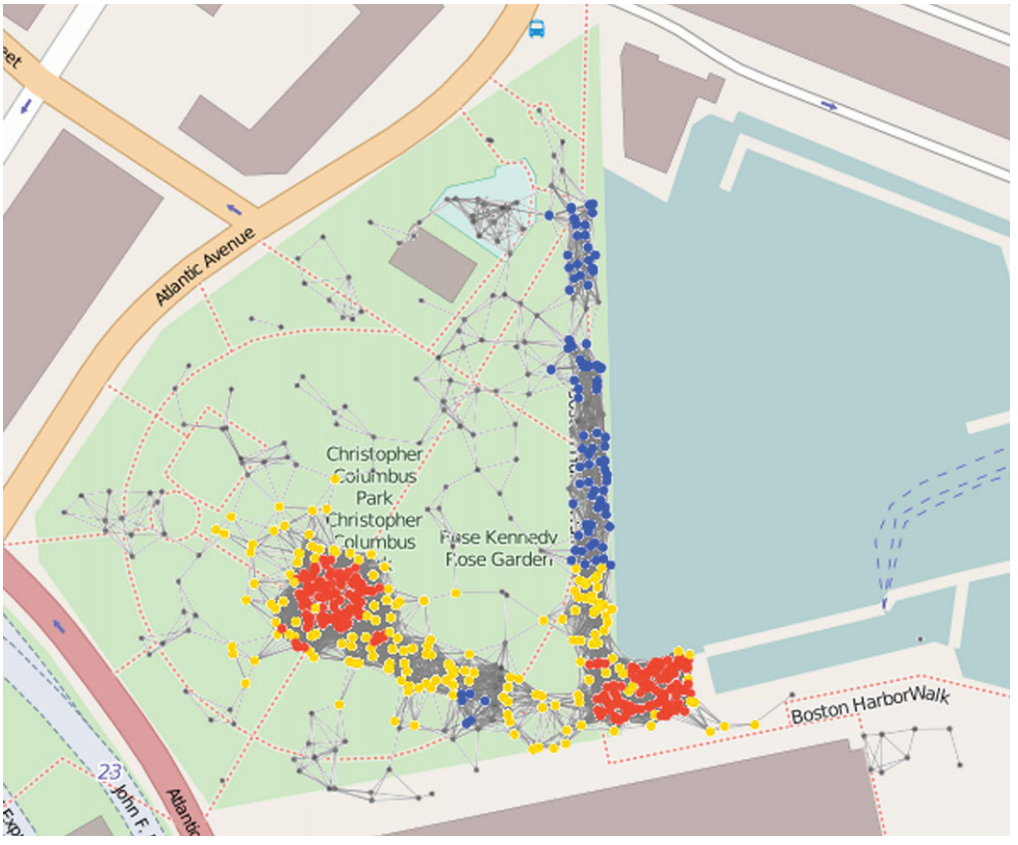

Figure 8. Snapshot of simulation of crowd management application (from [24]), implemented by composing 10 library functions based on an algebra of five building block operators and a number of simple mathematical and sensor built-in functions: dangerously crowded areas are marked in red, nearby areas being alerted in orange, dense but safe areas are blue, while network links and non-crowded devices are grey. (Online version in colour.)

running the crowd-management application: devices identify dangerous densities, alert those nearby, suggest routes to safety and navigate people around dense areas of crowd.

The application is self-stabilizing, resilient against many types of disruption, and can scale to operate on much larger or smaller networks. Despite the complexity and resilience of the application, its code is quite simple and terse. All four services are specified with a mere 18 lines of code (not counting comments and whitespace). This comprises 10 library calls, connected together with various built-in functions, expanding to a total of 18 building block operators (3 `if`, 9 `G`, 3 `C`, 2 `T` and 1 `S`).

This example illustrates the potential for aggregate programming to greatly enhance our ability to rapidly and reliably develop space–time computing applications. At present, of course, these are still quite early and experimental results, and there are significant challenges still to be overcome. One critical one is to strengthen the resilience properties considered: self-stabilization provides a guarantee of ultimately correct behaviour but is silent on the proportionality of disruption to size of perturbation. Something more like the notion of stability in linear and feedback systems needs to be extended to the more general computational context of aggregate programming, and the building blocks adjusted as necessary to comply with it and ensure that resilient systems also progress towards desired states with minimal ongoing disruption. The set of available library functions also needs to be increased to support a broader set of applications, and it will likely also be necessary to expand the set of building blocks to support functionality not yet covered, such as controlling the movement of devices.

6. Summary and conclusion

This paper has brought together a number of recent results to illustrate the state of the art in aggregate programming as a means for the effective programming and control of large networks of spatially embedded computing devices, as well as a roadmap for continuing to develop these techniques towards widespread practical application. This line of investigation is becoming increasingly necessary as space–time computing systems are continuing to rapidly increase in prevalence, driven by the ongoing decrease in cost and increase in capabilities of computing devices, as well as the diversification of computing devices into novel substrates such as smart materials and biological cells.

The ultimate goal of aggregate programming is to make the engineering of such distributed systems as routine as the engineering of individual computing devices. Moreover, many of the techniques developed for space–time programming appear to be equally applicable to any relatively sparsely connected networks, meaning they may also have more general use in improving the engineering of distributed systems.

A number of key open questions remain, however, needing further investigation before this vision can be resolved. At a theoretical level, there are important unresolved questions regarding the mapping between continuous and discrete models of device mobility, and on bounds for the quality with which continuous programs can be approximated by discrete networks. There is also an important question of how to define and achieve sufficient composable resilience properties, as well as how to approach security questions for decentralized aggregates. Assuming sufficient resilience and security can be guaranteed, it is expected that the set of building block algorithms will need to be extended in order to support broad domain-specific libraries, and these libraries must be developed in order for aggregate programming to be readily used in various different application domains. There is thus much work still to be done, but the potential rewards are high, particularly given the ever-increasing dependence of our civilization on complex cyber-physical infrastructure, and the need for this infrastructure to be extremely resilient despite ever-growing complexity and demands upon it.

Funding. This work is partially supported by the United States Air Force and the Defense Advanced Research Projects Agency (DARPA) under contract nos. FA8750-10-C-0242. The US 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 authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US Government. Approved for public release, distribution unlimited.

References

1. Beal J, Dulman S, Usbeck K, Viroli M, Correll N. 2013 Organizing the aggregate: languages for spatial computing. In *Formal and practical aspects of domain-specific languages: recent developments* (ed. M Mernik), pp. 436–501. Hershey, PA: IGI Global.
2. MacLennan B. 1990 Continuous spatial automata. Department of Computer Science Technical Report CS-90-121. Knoxville, TN: University of Tennessee.
3. MacLennan B. 1990 Field computation: a theoretical framework for massively parallel analog computation, parts I–IV. Department of Computer Science Technical Report CS-90-100. Knoxville, TN: University of Tennessee.
4. Margolus N. 1996 CAM-8: a computer architecture based on cellular automata. In *Pattern formation and lattice-gas automata* (eds AT Lawniczak, R Kapral), pp. 167–187. Providence, RI: American Mathematical Society.
5. Sklar E. 2007 NetLogo, a multi-agent simulation environment. *Artif. Life* **13**, 303–311. (doi:10.1162/artl.2007.13.3.303)
6. Ashley-Rollman MP, Goldstein SC, Lee P, Mowry TC, Pillai P. 2007 Meld: a declarative approach to programming ensembles. In *IEEE Int. Conf. on Intelligent Robots and Systems* (*IROS*), pp. 2794–2800. (doi:10.1109/IROS.2007.4399480).
7. Gilpin K, Kotay K, Rus D, Vasilescu I. 2008 Miche: modular shape formation by self-disassembly. *Int. J. Robotic Res.* **27**, 345–372. (doi:10.1177/0278364907085557)

8. Lasser C, Massar JP, Miney J, Dayton L. 1988 *Starlisp reference manual*. Cambridge, MA: Thinking Machines Corporation.
9. Madden S, Franklin M, Hellerstein J, Hong W. 2005 TinyDB: an acqusitional query processing system for sensor networks. *ACM Trans. Database Syst.* **30**, 122–173. (doi:10.1145/1061318.1061322).
10. Newton R, Welsh M. 2004 Region streams: functional macroprogramming for sensor networks. In *1st Int. Workshop on Data Management for Sensor Networks* (*DMSN*), pp. 78–87. New York, NY: ACM. (doi:10.1145/1052199.1052213)
11. Bachrach J, Beal J, McLurkin J. 2010 Composable continuous space programs for robotic swarms. *Neural Comput. Appl.* **19**, 825–847. (doi:10.1007/s00521-010-0382-8)
12. Giavitto JL, Godin C, Michel O, Prusinkiewicz P. 2002 *Computational models for integrative and developmental biology.* Technical report no. 72-2002. Evry, France: Univerité d'Evry.
13. Giavitto JL, Michel O, Cohen J, Spicher A. 2004 *Computation in space and space in computation.* Technical report no. 103-2004. Evry, France: Univerité d'Evry.
14. The Klavins Lab. 2012 *Gro: the cell programming language* (ed. E Klavins). Seattle, WA: University of Washington.
15. Beal J. 2005 Programming an amorphous computational medium. In *Unconventional programming paradigms* (eds JP Banatre, P Fradet, JL Giavitto, O Michel). Lecture Notes in Computer Science, vol. 3566, pp. 121–136. Berlin, Germany: Springer. (doi:10.1007/11527800_10)
16. Beal J. 2010 A basis set of operators for space–time computations. In *IEEE 4th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshop*, pp. 91–97. (doi:10.1109/SASOW.2010.21).
17. Apker TB, Potter MA. 2012 Robotic swarms as solids, liquids and gasses. In *AAAI Fall Symp. on Human Control of Bioinspired Swarms, Arlington, VA, 2–4 November 2012*.
18. Beal J, Usbeck K, Benyo B. 2013 On the evaluation of space-time functions. *Comput. J.* **56**, 1500–1517. (doi:10.1093/comjnl/bxs099)
19. Viroli M, Damiani F, Beal J. 2013 A calculus of computational fields. In *Advances in service-oriented and cloud computing* (eds C Canal, M Villari). Communications in Computer and Information Science, vol. 393, pp. 114–128. Berlin, Germany: Springer. (doi:10.1007/978-3-642-45364-9_11)
20. Lee TD, Yang CN. 1952 Statistical theory of equations of state and phase transitions. II. Lattice gas and Ising model. *Phys. Rev.* **87**, 410–419. (doi:10.1103/PhysRev.87.410)
21. Frisch U, Hasslacher B, Pomeau Y. 1986 Lattice-gas automata for the Navier–Stokes equation. *Phys. Rev. Lett.* **56**, 1505–1508. (doi:10.1103/PhysRevLett.56.1505)
22. Mamei M, Zambonelli F. 2006 Self-maintained distributed tuples for field-based coordination in dynamic networks. *Concurrency Comput. Pract. Exp.* **18**, 427–443. (doi:10.1145/967900.968000)
23. Montagna S, Viroli M, Fernandez-Marquez JL, DiMarzoSerugendo G, Zambonelli F. 2013 Injecting self-organisation into pervasive service ecosystems. *Mobile Netw. Appl.* **18**, 398–412. (doi:10.1007/s11036-012-0411-1)
24. Beal J, Viroli M. 2014 Building blocks for aggregate programming of self-organising applications. In *IEEE 8th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshop*, pp. 8–13. (doi:10.1109/SASOW.2014.6).
25. Pianini D, Viroli M, Zambonelli F, Ferscha A. 2014 HPC from a self-organisation perspective: the case of crowd steering at the urban scale. In *Int. Conf. on High Performance Computing Simulation* (*HPCS*), pp. 460–467. (doi:10.1109/HPCSim.2014.6903721)
26. Still GK. 2014 *Introduction to crowd science*. Boca Raton, FL: CRC Press.
27. Whitehouse K, Sharp C, Brewer E, Culler D. 2004 Hood: a neighbourhood abstraction for sensor networks. In *Proc. 2nd Int. Conf. on Mobile Systems, Applications, and Services*, pp. 99–110. New York, NY: ACM. (doi:10.1145/990064.990079)
28. Mamei M, Zambonelli F. 2009 Programming pervasive and mobile computing applications: the TOTA approach. *ACM Trans. Software Eng. Methodol.* **18**, 15. (doi:10.1145/1538942.1538945)
29. Gropp W, Lusk E, Skjellum A. 1994 *Using MPI: portable parallel programming with the message passing interface*. Cambridge, MA: MIT Press.
30. Zambonelli F *et al*. 2014 Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive Mobile Comput*. **17**, 236–252. (doi:10.1016/j.pmcj.2014.12.002)
31. Coore D. 1999 Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer. PhD thesis, MIT, Cambridge, MA, USA.

32. Nagpal R. 2001 Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics. PhD thesis, MIT, Cambridge, MA, USA.
33. Yamins D. 2007 A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems. PhD thesis, Harvard University, Cambridge, MA, USA.
34. Beal J, Bachrach J. 2006 Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intell. Syst.* **21**, 10–19. (doi:10.1109/MIS.2006.29)
35. Pianini D, Beal J, Viroli M. In press. Practical aggregate programming with PROTELIS. In *ACM Symp. on Applied Computing* (*SAC 2015*).
36. Church A. 1932 A set of postulates for the foundation of logic. *Ann. Math.* **33**, 346–366. (doi:10.2307/1968337)
37. Milner R. 1999 *Communicating and mobile systems: the pi-calculus*. Cambridge, UK: Cambridge University Press.
38. Igarashi A, Pierce BC, Wadler P. 2001 Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* **23**, 396–450. (doi:10.1145/503502.503505)
39. Viroli M, Beal J, Usbeck K. 2013 Operational semantics of Proto. *Sci. Comput. Program.* **78**, 633–656. (doi:10.1016/j.scico.2012.12.003)
40. Beal J, Viroli M, Damiani F. 2014 Towards a unified model of spatial computing. In *7th Spatial Computing Workshop* (*SCW 2014*), Paris, France, 5–9 May 2014, pp. 59–64.
41. Viroli M, Damiani F. 2014 A calculus of self-stabilising computational fields. In *Coordination models and languages* (eds E Kühn, R Pugliese). Lecture Notes in Computer Science, vol. 8459, pp. 163–178. Berlin, Germany: Springer. (doi:10.1007/978-3-662-43376-8_11)
42. Damiani F, Viroli M, Pianini D, Beal J. In press. Code mobility meets self-organisation: a higher-order calculus of computational fields. In *35th IFIP Int. Conf. on Formal Techniques for Distributed Objects, Components and Systems*.
43. Fernandez-Marquez J, Marzo Serugendo G, Montagna S, Viroli M, Arcos J. 2013 Description and composition of bio-inspired design patterns: a complete overview. *Nat. Comput.* **12**, 43–67. (doi:10.1007/s11047-012-9324-y)