Self-Adaptation to Device Distribution Changes

Self-adaptation to Device Distribution Changes

Jacob Beal*, Mirko Viroli†, Danilo Pianini†, and Ferruccio Damiani‡
*Raytheon BBN Technologies, USA; Email: jakebeal@bbn.com
†Università di Bologna, Italy; Email: {mirko.viroli,danilo.pianini}@unibo.it
‡Università di Torino, Italy Email: damiani@di.unito.it

***Abstract*—A key problem when coordinating the behaviour of devices in situated networks (e.g., pervasive computing, smart cities, Internet of Things, wireless sensor networks) is adaptation to changes impacting network topology, density, and heterogeneity. Computational goals for such systems are often expressed in terms of geometric properties of the continuous environment in which the devices are situated, and the results of resilient computations should depend primarily on that continuous environment, rather than the particulars of how devices happen to be distributed through it. In this paper, we identify a new property of distributed algorithms, *eventual consistency*, which guarantees that computation self-stabilizes to a final state that approximates a predictable limit as the density and speed of devices increases. We then identify a large class of programs that are eventually consistent, building on prior results on the field calculus computational model [1], [2] to identify a class of self-stabilizing programs. Finally, we confirm through simulation of pervasive network scenarios that eventually consistent programs from this class can provide resilient behavior where programs that are only self-stabilizing fail badly.**

I. Introduction

A number of research and development thrusts, including pervasive computing, smart cities, and the "Internet of Things," all aim toward an environment increasingly densely saturated with pervasive and interconnected devices. This new sort of *situated network* poses new engineering challenges, particularly in the area of distributed coordination: many interactions need to be opportunistic, context-dependent, and based on physical proximity, and their large scale implies they must be adaptive and resilient to nearly continual faults and changes in the network. Situated networks are also highly heterogeneous in distribution, due to the underlying heterogeneity in human activity and environmental structure that drives the deployment of network devices. For example, a city center is likely to host a high density of cars, traffic sensors, and smart signage, while a country road has very few by comparison. Likewise, an office plaza may host many devices by day, few at night, and massive numbers during sporting events or festivals.

Developing applications for such environments can be extremely difficult, as conventional development methods require that a programmer simultaneously address networking protocols, coordination mechanisms, and the application itself. This tends to lead to fragile applications, particularly due to the difficulty of adequate testing for resilience.

Aggregate programming [1] simplifies the problem by factoring distributed systems development into several layers: *field calculus* maps between aggregate-level computations and local interactions between individual devices to implement those computations [3], [4], systems of composable "building block" operators implemented in field calculus provide resilience and scalability guarantees [5], [2], and domain-general and domain-specific APIs built using building block operators provide a programatic interface for construction of complex networked services and applications [5], [2].

Here we focus on resilience and adaptation guarantees in the "building block" layer: prior work has established self-stabilizing programs, but results of those programs may depend sensitively on details of how devices are distributed in space and time. We thus introduce a new and stronger property, *eventual consistency*, which guarantees that a computation not only self-stabilizes, but also that its values are a good approximation of executing a computation on the continuous environment in which the network is situated, thereby effectively giving a notion of "independence" of computation from the underlying network details.

Following a brief review of related work in Section II, we provide a formal model of continuum computation and use this to define *eventual consistency* in Section III. Section IV identifies a subset of the self-stabilizing calculus presented in [2] that also provides eventual consistency. Finally, Section V demonstrates the breadth of this sublanguage and empirically confirms the value of eventual consistency in simulation of pervasive network scenarios, and Section VI summarizes the contributions of this paper and discusses future work.

II. Related Work

A wide range of aggregate programming methods have been developed for engineering situated networks. A thorough review may be found in [6], which identifies four main approaches to aggregate programming: simplifying the interface for programming individual devices and their interactions (e.g., TOTA [7], MELD [8], $\sigma\tau$-Linda [9]), creation of geometric and topological patterns (e.g., Origami Shape Language [10], Growing Point Language [11], or Yamins' universal patterns [12]) summarizing and streaming information over regions of space and time (e.g., TinyDB [13], Regiment [14]), and general purpose space-time computing models (e.g., MGS [15], StarLisp [16], field calculus [3]). The general purpose models, and in particular field calculus [3], [4] have been the basis for a layered approach to building distributed adaptive systems as presented in our previous work [1], [2].

More generally, identifying robust (e.g., self-adapting or self-stabilizing) algorithms for distributed systems is a long investigated problem [17] which is part of the general challenge of devising sound techniques for engineering self-organising applications. To the best of our knowledge, however, the only works aiming at a proof of self-stabilization for an entire class of computational field algorithms are [18] and [2]. In particular, these works consider deterministic self-stabilization, in which the system eventually reaches a stable state completely determined by the environment (network topology and sensor values), whenever the environment does not change for sufficient amount of time. As argued in [19], there is expected to be a whole catalogue of self-organization patterns susceptible to similar approaches. These prior works have a large range of expressiveness, but are unable to tie the properties they investigate to a continuous environment or to consider similarity between networks with different topologies, more challenging properties that eventual consistency addresses.

III. Eventual Consistency

Self-stabilization is a well-established theoretical property of distributed algorithms [17]: under the standard definition, a system self-stabilizes if it is guaranteed to converge from any arbitrary initial state to some state with a defined set of "correct" properties. We now introduce a new property, *eventual consistency*, that goes one step further tying state to the space in which the network is situated: intuitively, a system is eventually consistent if the state it converges to is set by the continuous environment rather that the particulars of how devices are distributed through that environment. This model fits well with situated networks, since many computations are more concerned with the environment in which devices are situated rather than the particulars of individual devices, and can hence be naturally described in geometric terms, e.g., distances, regions, information flow.

We will develop this concept in three stages: first, we review how situated networks can be viewed as an approximation of a continuous environment, per [20], [21], which then forms a basis for both discrete and continuous models of space-time computation, per [22], [23]. We then use these concepts to define eventual consistency and examine how this definition implies resilience to perturbation in the density and arrangement of devices in a situated network.

A. Networks as Approximations of a Continuum

Many physically situated networks perform computations that can be naturally described in terms of the physical space through which the devices comprising the network are distributed. In this case, as observed in [20] and [21], a network can be viewed as a discrete approximation of the continuous physical space and programmed accordingly.[1]

[1]Note that there are a number of other continuum computation models that might be considered as well, including hybrid automata [24] and continuous spatial automata [25], but we have preferred the model that directly inspired field calculus and thus maps most directly onto it.

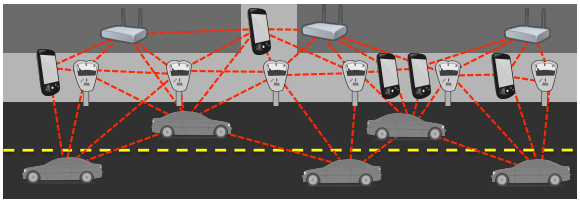


Fig. 1. Example of a situated network in a street environment; the walkable spaces of the street form a manifold.

Under the continuous model developed in those papers, computation takes place on a Riemannian manifold M with both space and time dimensions. A manifold is a space that is locally Euclidean, but may have more complex structure over a longer range. Riemannian manifolds also support familiar geometric constructs like angles, lengths, curvature, integrals and derivatives. This allows communication and mobility constraints to be embedded in the manifold's geometry, measuring distance through the manifold rather than using absolute (e.g., latitude/longitude) coordinates. For example, the walkable spaces of a street (e.g., Figure 1) form a Riemannian manifold in which the shortest distance between locations goes along sidewalks, roads, and plazas, rather than through the walls of buildings.

Communication in a continuous space is modeled as a bound c on the speed at which information can propagate. A set of concepts and terminology from physics may then be borrowed to describe the space-time relations of a manifold [26]. To wit, a point $m \in M$ is termed an "event," denoting its interest as both a spatial and temporal location, and the manifold may be partitioned with respect to m in space and time (see Figure 2):

- Events that information can go to or from at exactly c are *simultaneous* with m.
- The set of events that can be reached from m moving slower than c, denoted $T^+(m)$, is the *time-like future* of m, while events whose information reaches m moving slower than c are its *time-like history* $T^-(m)$.
- All other events, which cannot share information because it would need to move faster than c, have *space-like separation* from m and no natural order.

A *device* d can be any one-dimensional curve in the manifold with purely time-like relations between the points.[2] The mathematical analysis in this paper also makes the simplifying assumptions that the manifold is finite in diameter and devices do not move.[3] Finally, a *spatial section* S_M is a distributed snapshot of state without a global notion of time, formally, any set S_M such that $T^+(S_M) \cup T^-(S_M) = M - S_M$.

Figure 2 gives a concrete pervasive computing example of these concepts in terms of several devices distributed in

[2]This is analogous to the physics notion of a world-line, i.e., devices have a history in time, but only exist at one point in space at time.

[3]We defer mobile devices to future work, though in practice these results typically apply well to devices moving much more slowly than c.

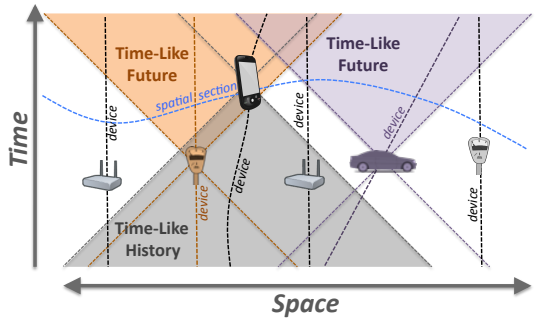


Fig. 2. Example of continuous space-time relations between six devices distributed along a street: wireless access points and meters are stationary, while the car moves steadily and the phone stops and starts twice.

one dimension along a city street. Each device is represented by a time-like trajectory indicating its position over time: the access points and parking meters are stationary (vertical lines) while the car moves steadily and the phone stops and starts twice. The event marked by the phone on its trajectory is in the time-like future of the marked event on the orange meter's history, meaning it can be affected by the meter's state at that time. However, it is space-like separated from the marked event on the car's history, meaning it only has access to older information about the car. Finally, the blue line marks one of many possible spatial section "snapshots" that can separate all of space-time into a "strict future" and "strict past."

B. Computations Across Space and Time

Standard event-based models of distributed computation (e.g., [27]) cannot be applied to continuous spaces, as any manifold contains an uncountably infinite number of events, and thus the events of the computation cannot be placed in a countable sequence. Instead, computation on manifolds may be defined in terms of fields, per [21], [22], [23]:

Definition 1 (Continuous Computational Field) *A field is a function $f : M \to \mathbb{V}$ that maps every event m in a Riemannian manifold M to some data value in $\mathbb{V}$.*

Discrete computational fields (e.g., actions taken by real devices in the execution of a distributed algorithm) may be defined likewise, except that the domain is limited to a discrete subset of events $D \subset M$.[4]

For example, consider a situated network in a street environment such as is shown in Figure 1. Two examples of continuous computational fields are shown in Figure 3(a): the top a real-valued field of temperatures (shading blue to red from lowest to highest), the bottom a Boolean-valued field indicating where the temperature is greater than 20°C (green for true, orange for false). Figure 3(b) shows examples of discrete fields: these fields also show temperature and comparison (using the same color scheme), but contain only the values located at individual devices in the network, rather than across the whole region of space.

A space-time computation C is a higher-order function mapping an input field to a corresponding field of values:

Definition 2 (Space-Time Computation) *Let $\mathbb{F}_\mathbb{V}$ be the set of fields with range $\mathbb{V}$, a computation C is a function $C : \mathbb{F}_\mathbb{V} \to \mathbb{F}_\mathbb{V}$, where the domain of the output field is always identical to the domain of the input field.*

In other words, a computation takes an *evaluation environment* field, whose domain defines the scope over which the computation executes and whose values are all of the environmental state that can affect its outcome (e.g., sensor readings). At every point of space and time in the execution scope, some output value is produced. For example, Figure 3(a) and 3(b) show examples of continuous and discrete computation, in which the field of temperatures is passed through a function that compares to 20°C to compute a Boolean field that indicates the locations of higher temperatures.

Such space-time computations can be specified by functional composition of a basis set of operators:

Definition 3 (Space-Time Operator) *A space-time operator is a function $o : \mathbb{F}_\mathbb{V} \times \mathbb{F}_\mathbb{V}^k \to \mathbb{F}_\mathbb{V}$ taking an evaluation environment and zero or more additional fields as inputs and producing a field as output.*

This is much like the definition of a computation, except that the domains of the fields may differ.

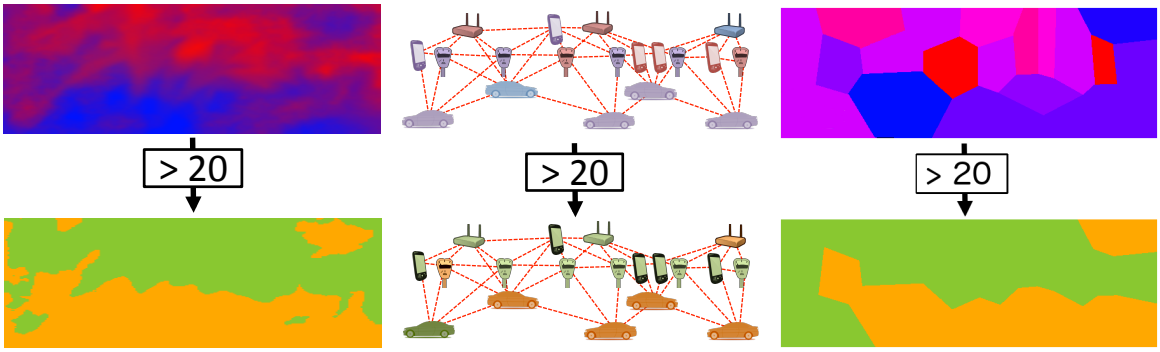
Considering operators as constructs of a programming language, we can see a *space-time program* as any functional composition of operator instances to form a computation, such that the domains and ranges of the output are well-defined for all possible values of the inputs.[5] Note that this includes recursive composition, e.g., via lambda calculus, so programs are potentially universal, per [22].

For example, the space-time program in Figure 3 might be defined as a composition of three operators: one returning the field of environment temperatures, another returning a constant-valued field of 20, and a third comparing its two inputs point-wise to find at which events the value of the first input is greater than the value of the second.

Note that for clarity in describing programs, we will abuse terminology; in the context of a program, a "field" is not actually the mathematical object itself, but the input or output of an operator instance, which takes on a field value when evaluated in the context of an environment.

[4]Defining discrete execution as a manifold subspace (not an abstract graph) preserves the geometric relationship of network to environment.

[5]Some notes on technicalities: 1) Any well-defined composition of operators is itself an operator. 2) Complete programs have no inputs except the environment. Any composition of operators that requires inputs (e.g., function definitions) may, however, be transformed into an equivalent program by "currying" the inputs to instead be supplied by the environment and adjoining a special "no value" value for events in the domain of the environment but not the input.



(a) Continuous (b) Discrete (c) ϵ-approximation

Fig. 3. On the situated network in a street environment in Figure 1: a continuous computation of a temperature threshold (b) is approximated by the discrete network of devices (c), producing the ϵ-approximation shown in (d).

C. Relating Continuous and Discrete Computing

We can now consider what it means for the results of a situated network computation to be determined by its continuous environment. Our basic approach will be to define the "ideal" outcome of a space-time program as its results when applied to a continuous environment, then compare this with the results for a discrete network of devices situated in that same environment.[6] We can compare a continuous field to a corresponding discrete field by mapping the domain of the continuous field to the values of the nearest points in the discrete field:

Definition 4 (ϵ-approximation) *Let $D_\epsilon \subset M$ be a discrete set such that every event $m \in M$ is within distance ϵ of some event in D_ϵ. The ϵ-approximation of field $f : D_\epsilon \to \mathbb{V}$ is a field mapping every point in M to the value of f at the nearest point in D_ϵ (choosing arbitrarily for equidistant points).*

An example is shown in Figure 3(c), which illustrates the ϵ-approximation of the fields of the discrete computation in Figure 3(b) on the manifold of the environment illustrated in Figure 1. Notice that this is a coarse approximation of the continuous computation in Figure 3(a). For this example, it is readily apparent that the more devices there are, the more the ϵ-approximation would look like the ideal continuous computation. This is the notion of field approximation:

Definition 5 (Field Approximation) *A field $f : M \to \mathbb{V}$ is approximated by a countable sequence of ϵ_i-approximations f_i of manifolds M_i, with $\epsilon_i < \epsilon_{i-1}$ and ϵ_i going to zero, if both the following hold:*

$$\lim_{i\to\infty} |(M \cup M_i) - (M \cap M_i)| = 0$$

$$\lim_{i\to\infty} \int_{M\cap M_i} |f - f_i| = 0$$

In other words, a sequence of increasingly fine discrete sets approximates a continuous field if both the manifolds and the values assigned over them by the fields tend toward identical.[7] We can then define a *consistent program* as one where the approximation relationship always holds:

Definition 6 (Consistent Program) *Let $\mathtt{P}$ be a space-time program, e be an evaluation environment, and e_i a countable sequence of ϵ_i-approximations that approximate field e. Program $\mathtt{P}$ is* consistent *if $\mathtt{P}(e_i)$ approximates $\mathtt{P}(e)$ for every e_i and e.*

Since information may move at different speeds in different discrete approximations, most programs involving communication are not consistent. A program converging to a steady state, however, may be consistent after that point:

Definition 7 (Eventually Consistent Program) *Consider a causal program $\mathtt{P}$ evaluated on domain M. Program $\mathtt{P}$ is* eventually consistent *if, for any evaluation environment e with a spatial section S_M such that the values of e do not change at any device in the time-like future $T^+(S_M)$, there is always some spatial section S'_M such that $\mathtt{P}$ is consistent on the time-like future $T^+(S'_M)$*

In other words: if the inputs ever converge, then the outputs eventually converge as well, and are consistent thereafter. For example, the temperature program in Figure 3 is both consistent and eventually consistent. A "gossip" algorithm that uses its output to compute whether any location had seen a high temperature, however, would only be eventually consistent, since the speed that gossip can propagate information can be affected by the particulars of discretization.

The value of eventual consistency is that it implies certain types of resilience. First, eventual consistency implies that a computation is not particularly sensitive to precise locations of devices, since the values must converge for all e_i sequences. Second, results can only improve (asymptotically)

[6]Note that we consider only causal computations; acausal computations are well-defined but cannot generally be implemented because they use information from the future; see [21] for details.

[7]Note: the reason to use a sequence of potentially different manifolds M_i is because program branches can create subspaces dynamically, and these necessarily depend on the details of approximation.

```
l ::= 𝕍                                              ;; Literals
e ::= x | l | (b e1 ... en) | (f e1 ... en)          ;; expression
    | (rep x w e) | (nbr e) | (if e e e)             ;; special constructs
w ::= x | l                                          ;; variable or value
F ::= (def f(x1 ... xn) e)                           ;; function
P ::= F1 ... Fn e                                    ;; program
```

(a) Syntax of Field Calculus

```
l ::= 𝔹 | ℤ | ℝ | ℬ                                  ;; Literals
b ::= m | mux | <                                    ;; local operators
e ::= x | l | (b e1 ... en) | (f e1 ... en)          ;; expression
    | (sense ℤ+)                                     ;; sensor
    | (if e e e) | (GPI e e e e)                     ;; special construct
F ::= (def f(x1 ... xn) e)                           ;; function
P ::= F1 ... Fn e                                    ;; program
```

(b) Restriction to GPI-calculus sub-language

Fig. 4. Field calculus [3], [4] is a minimal computational calculus that does not ensure eventual consistency. GPI-calculus is a restriction to a sub-language of eventually consistent programs.

as the number of devices in the network increases. Third, combining location insensitivity and improvement with density, eventually consistent computations should also typically be quite tolerant of network heterogeneity. Furthermore, when a computation is not eventually consistent, the manner in which is it is not consistent is likely to reveal system vulnerabilities that need to be considered even when a situated network is not expected to be particularly dense or fast changing.

IV. Eventually Consistent Language

Having established eventual consistency as a desirable property, we now provide a methodology for the construction of systems with this property, by identifying an expressive system of programming constructs, such that any program comprised solely of such constructs is guaranteed to be eventually consistent. In particular, our approach begins with the self-stabilizing sub-language of field calculus presented in [2]. Following a brief review of field calculus and its self-stabilizing sub-language, we then analyze how programs that are not eventually consistent can have behavior that is extremely sensitive to small changes in the arrangement of devices in space. Using this analysis, we then identify a highly expressive restriction of the self-stabilizing sub-language that contains only eventually consistent programs, called GPI-calculus.

A. Field Calculus

Field calculus [3], [4] is a minimal universal language, in which every expression specifies a space-time program, as defined in Section III-B. That is, a field calculus program takes a field specifying the evaluation environment as input and outputs a field of results. Importantly, field calculus is universal (meaning it can express any physically realizable computation), small enough to be tractable to analyze formally, and can be applied to both continuous and discrete fields [22], which means that it is a good framework for investigating eventual consistency.

Field calculus programs are specified using the syntax in Figure 4(a): each program is either a literal `l`, defining a field that maps to the same data value everywhere (e.g., `3` is a field whose value at every point is 3), or a composition of the following constructs:

- *Built-in operators:* A built-in operator $(\texttt{b}\ \texttt{e}_1 \ldots \texttt{e}_n)$ determines the value of its output field at event m only from the values of the environment e and input fields $\texttt{e}_1, \texttt{e}_2, \ldots$ at m. The built-in operators can range over any such functions, including addition, comparison, sensors, actuators, etc.
- *Function definition and call:* New functions can be defined Lisp-style with expressions of the form $(\texttt{def}\ \texttt{f}(\texttt{x}_1 \ldots \texttt{x}_n)\ \texttt{e})$ and called with expressions of the form $(\texttt{f}\ \texttt{e}_1\ \ldots\ \texttt{e}_n)$.
- *Time evolution:* Program state is initialized and changed over time by a "repeat" construct $(\texttt{rep}\ \texttt{x}\ \texttt{w}\ \texttt{e})$, initializing `x` to a literal or variable and updating (non-synchronously) by computing `e` against its prior value.
- *Neighborhood values:* At each event, expression $(\texttt{nbr}\ \texttt{e})$ constructs a sub-field mapping neighboring devices to their most recent value of `e`. These sub-fields can then be manipulated and summarized with built-in operators. For example, `(min-hood (nbr e))` maps each device to the minimum value of `e` amongst its neighbors (excluding itself).
- *Domain restriction*: $(\texttt{if}\ \texttt{e}_0\ \texttt{e}_1\ \texttt{e}_2)$ computes expressions in subspaces, preventing interference between the two sub-computations: $\texttt{e}_1$ is computed where Boolean $\texttt{e}_0$ is true, $\texttt{e}_2$ where it is false.

A field calculus program is then a set of function definitions followed by an expression to be evaluated. Thus the example in Section III-C of an eventually consistent "gossip" algorithm that computes whether any location has ever experienced a high temperature, can be implemented:

```
(def gossip-ever (value)
  (rep ever
    false
    (or value (any-hood (nbr ever)))))
(gossip-ever (> (temperature) 20))
```

Here, the gossip process is defined using `def`, with a combination of time evolution and neighborhood value constructs. The program remembers if there has ever been a high value with the Boolean field `ever`. This switches to *true* at an event in two cases, joined with built-in `or`: either the input field `value` is *true* or else information arrives that some neighbor has switched to *true*: `(nbr ever)` collects values from neighbors and `any-hood` returns *true* if its input has a *true* value for any neighbor.

Because field calculus is universal, it can express any program, including non-resilient programs. A sub-language was identified in [2], however, where self-stabilization can be guaranteed by using `nbr` and `rep` constructs only in three patterns, which may be thought of roughly as spreading, folding, and bounded monotonic change, and

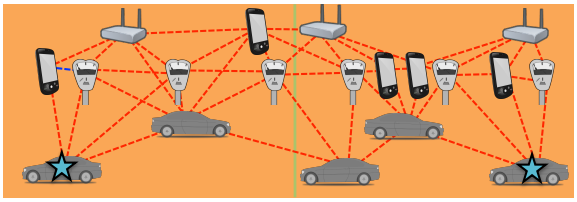


Fig. 5. Finding a bisector is fragile because it is sensitive to device positions. For example, the set of bisecting locations (green line) for two cars (blue stars) might or might not actually include any devices.

which cover a large number of self-stabilizing algorithms.

B. From Consistency Failures to Fragility

Even if a function is self-stabilizing, it may not be eventually consistent. Let us then examine how consistency failures emerge in field calculus. Three modes of consistency failure arise directly from constructs that, while useful, can also be readily used to create programs that can never converge to a well-defined behavior. First, recursion can create consistency failures, since there are many ways to arrange a recursion that grows in depth with density and thus does not converge. Second, interactions between neighboring devices can lead to consistency problems due to implicit dependence on the distribution of devices: for example, a measurement that counts hops will not converge if increasing density of devices leads to paths consisting of more small hops. State constructs can also create in programs that implicitly rely on the time between events, but the self-stabilizing sublanguage in [2] prevents this by means of the restricted patterns it allows for use of state constructs. Following a similar strategy to ensure eventual consistency, we will thus prohibit recursion and restrict use of neighbor to a pattern that can be guaranteed safe.

More subtlely, many computations converge but are extremely sensitive to individual devices. A good example is one of the most widely used self-stabilizing distributed algorithms, finding the distance to a source region:

```
(def distance-to (source)
  (rep d
    infinity
    (mux source 0
      (min-hood (+ (nbr d) (nbr-range))))))
```

This finds distance by incremental application of the triangle inequality, using built-in functions +, for pointwise addition, and `mux`, which multiplexes between its second and third inputs, returning the second where the first is *true* and the third elsewhere.

Although `distance-to` always converges to an approximable output, programs incorporating it may not be eventually consistent because the value of the output may be greatly affected by individual points in the `source` field. Consider, for example, a `source` field where only one point is *true*: an ϵ-approximation containing that point has only finite values, while one without that point has infinity everywhere. Thus, it is possible to construct sequences that do not converge, because they alternate between including and not including the critical point.

Although this example may seem extreme, it is easy to accidentally create such critical dependencies. For example, a simple bisecting boundary computation:

```
(def bisector (point-1 point-2)
  (= (distance-to point-1) (distance-to point-2)))
```

creates a field that is *false* except at an infinitely thin boundary of *true* values (Figure 5). Fed to a program sensitive to such sets, such as `distance-to`, this can result in arbitrarily unpredictable behavior from a distributed algorithm.

This is not a special case related to `distance-to`, but a deeper conflict for situated distributed algorithms, between the discrete values commonly used in algorithms (e.g., Booleans, branches, state machines) and the continuous space-time environment in which devices are embedded. In particular, any non-trivial field with a discrete range cannot be continuous, meaning that it either is itself not approximable or else contains some measure-zero boundary region that, if handled badly, can generate unpredictable behavior (as in the bisector example).

C. Restriction of Field Calculus: GPI-calculus

As [2] restricted field calculus to obtain a sub-language of self-stabilizing programs, we now further restrict the self-stabilizing sub-language to a sub-language of eventually consistent programs, which we call GPI-calculus, whose syntax is shown in Figure 4(b). As we find eliminating every problematic program element to be too limiting, this sub-language accepts boundary elements, but dynamically marks them to contain their effects. In particular:

- The possible data values are restricted to only Booleans ($\mathbb{B}$), integers ($\mathbb{Z}$), real numbers ($\mathbb{R}$), and a unique value $\mathcal{B}$ denoting a possibly problematic boundary between value regions.[8]
- Built-in operators are restricted to the elements `m`, `mux`, `<`, and `sense` (all described below).
- `if` allows $\mathcal{B}$ as a third value for its first input, in addition to Boolean true and false; for those points mapping to $\mathcal{B}$, the output also maps to $\mathcal{B}$.[9]
- State and communication are only available indirectly through a new operator, `GPI` (described below), which is a restriction of the spreading pattern in [2].
- Recursion is prohibited.

Boundary-Aware Built-In Functions: The built-in operators in GPI-calculus are close relatives of standard mathematical and sensor functions; the only difference is that they also interact with the boundary value:

- `m` is any strictly continuous mathematical function

[8]Note that restricting to the more computationally tractable rationals $\mathbb{Q}$ does not affect measure-zero fragility or its solution via boundary values.

[9]Not a semantic change: syntactic sugar on two nested `if` statements.

(e.g., addition, multiplication, logarithm, sine[10]), extended to have output $\mathcal{B}$ if any input is $\mathcal{B}$. Additionally, for any m that can map integers to integers (e.g., addition), the output has integer type iff all inputs have integer type; otherwise the output is a real.

- `mux` is the piecewise multiplexer function: when its first input is *true*, it returns the second input; if it is *false*, it returns the third input; if it is $\mathcal{B}$, it returns $\mathcal{B}$.
- `<` compares two numerical inputs, returning *true* if the first is less and *false* if the second is less. If they are equal, then the result depends on type: if both are integers the result is *false*; if either is a real it is $\mathcal{B}$. Finally, if either input is $\mathcal{B}$, then the result is also $\mathcal{B}$.
- (`sense` k) returns the kth value in the environment state (assumed to be a tuple), where k is the positive integer literal given as its input.

Composition makes this apparently limited set of operations actually generate a much wider set. E.g., `mux` is sufficient to implement all logical operations:

```
(def and (a b) (mux a b false))
(def or  (a b) (mux a true b))
(def not (x)   (mux x false true))
```

The `GPI` Operator: Key to distributed computation in the restricted language is the new operator `GPI`, a "gradient-following path integral," which we define as a field calculus function similar to operator G in [2]:

```
(def GPI (source initial density integrand)
 (if (<= density 0)
  B // Metric ill-defined if density non-positive
  (2nd
   (rep distance-integral
    (tuple infinity initial) // Initial value
    (mux source
     (tuple 0 initial)// Source is distance zero, initial value
     (min-hood' // Minimize lexicographically over non-self nbrs
      (+ (nbr distance-integral)
       (* (nbr-range) // Scalar multiplication of tuple
        (tuple (mean density (nbr density))
         (mean integrand (nbr integrand)))))))))))
```

Here, in addition to the previously discussed built-in operators, we also use `tuple`, which creates a k-tuple of its inputs, `2nd`, which accesses the second value of a tuple, and `mean`, which finds the average of its inputs. We also use a slightly modified version of the usual field-calculus `min-hood` operator, designated as `min-hood'`, which returns $\mathcal{B}$ if the minimal value for the first tuple element is held by more than one device.

The `GPI` operator thus performs two tasks simultaneously. First `GPI` computes a field of shortest-path distances to a `source` region. This distance is "stretched" proportional to a scalar field `density` (representing e.g., crowd density slowing movements, hazards increasing danger of movement). Furthermore, the `min-hood'` operator ensures that all points in the distance field with more than one shortest path are $\mathcal{B}$. Second, `GPI` computes a path integral of the scalar field `integrand` following the gradient of the distance field upward away from the source, starting at the scalar value `initial` in the source region. The function definition binds these together via a tuple and lexicographic minimization, such that the value added to the line integral at each device is taken from the neighbor on the (sole) minimal path to the source.

Importantly, just as with G in [2], the `GPI` operation subsumes a number of useful and frequently used computations. For example, an eventually consistent version of `distance-to` can be implemented as:

```
(def distance-to (source)
  (GPI source 0 1 1))
```

GPI-calculus is Eventually Consistent: With these restrictions, well-written programs will ensure eventual consistency by effectively excluding a minuscule (often empty) set of devices from certain computations. Poorly written programs (e.g., `(= (sqrt 2) (sqrt 2))`) may still contaminate large areas with $\mathcal{B}$ values, but will still converge—just not to a particularly useful result.

Theorem 1 (Eventual Consistency of GPI-calculus) *GPI-calculus programs are eventually consistent for all environments e that are continuous on $e^{-1}(\mathbb{V} - \mathcal{B})$.*

We present only a sketch of this proof here for reasons of length: First, we consider any set of operators that are eventually consistent and are continuity preserving, in the sense that when there is a spatial section S_M such that environment e and inputs f_i are continuous on $e^{-1}(\mathbb{V} - \mathcal{B}) \cap T^+(S_M)$ and $f_i^{-1}(\mathbb{V} - \mathcal{B}) \cap T^+(S_M)$, then there is some other S'_M such that their output f_o is continuous on $f_o^{-1}(\mathbb{V} - \mathcal{B}) \cap T^+(S'_M)$. All finite compositions of such operators have the same properties of eventual consistency and continuity preservation, since adding a single operator doesn't affect these properties. Each operator in GPI-calculus can then be shown to be eventually consistent and continuity preserving, in a lengthy but not particularly complex set of reasoning, the most complex of which is for GPI. Finally, we show that all GPI-calculus programs are equivalent to finite compositions of operators, which implies that all such programs are eventually consistent.

Thus, if a program is evaluated in a "well-behaved" environment, its results are predictable and resilient to scale and positioning of devices. Having proved this, in the next section we explore the breadth of applications that can be addressed by GPI-calculus and demonstrate its consistency properties empirically in simulation.

V. Validation and Applications

We now validate the predictions of eventual consistency and demonstrate some of the breadth of applications that can be expressed to our sub-language of eventually consistent programs. As the `GPI` operation is a restricted version of the G information spreading operator from [2],

[10]This also includes construction and referencing of tuples, which can be used to implement data structures, as well as higher order functions such as map and reduce. Some simple functions are excluded, however, such as division, which is discontinuous when the denominator is zero.

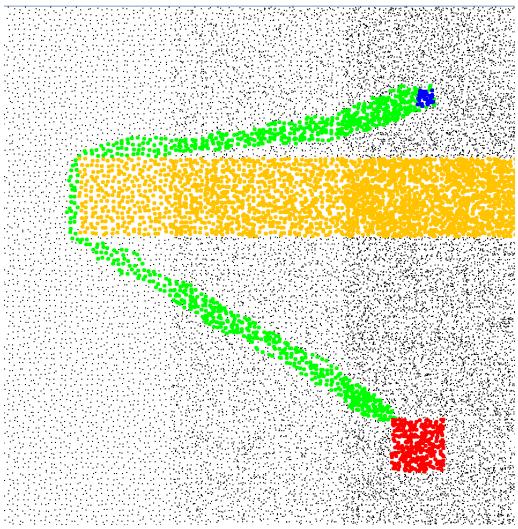
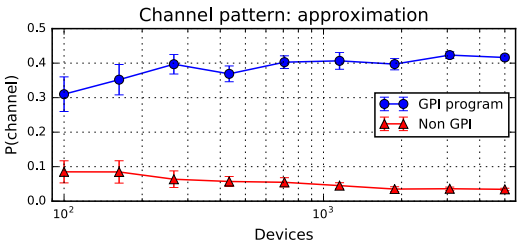
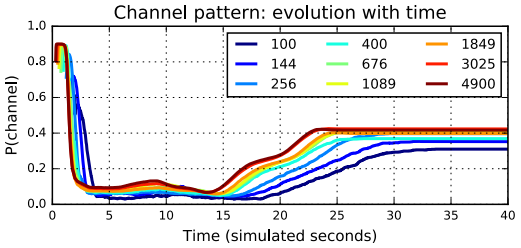


Fig. 6. Channel pattern deployed on a wireless sensor network with 20,000 devices deployed at heterogeneous density. Blue devices send data to red along the channel (green), avoiding the obstacle (yellow).



(a) GPI vs. non-GPI convergence



(b) Convergence vs. time

Fig. 7. Simulation of wireless sensor network scenario confirms the analytical results for GPI-calculus, showing convergence with respect to both number of devices (a) and time (b). Number of devices shows mean and ± 1 standard deviation for both the GPI (blue) and non-GPI (red) versions. Time graphs show mean of GPI version only, colored by number of devices shading from deep blue (100) to dark red (5000).

GPI-calculus applications are those based on information spreading and local computation. We present several such common self-organization patterns, along with accompanying application scenarios in wireless sensor networks and urban traffic steering. Simulations of these scenarios and comparison of GPI-calculus algorithms with similar algorithms that lack eventual consistency confirm the consistency result and its value for constructing coordination behaviors resilient to changes in network density and scale. Note that we do not attempt to cover the breadth of possible alternatives: rather, these comparisons show the difficulties that can arise if eventual consistency is left to programmers rather than being guaranteed by the framework.

A. Distance-Based Patterns

As previously noted, distributed distance calculation can be implemented with a simple `GPI` call. Another common pattern is broadcast from a source, which can be defined:

```
(def broadcast (source value)
  (GPI source value 1 0))
```

Here, `GPI` shifts the initial `value` outward by integrating 0 along the path, so that the value remains unchanged, thus producing a broadcast (or more generally, a map from each devices to the nearest source device's `value`).

These functions can then be composed into higher-level patterns. For example, a `channel`, useful for tasks like corridor routing, can be implemented:

```
(def channel (a b w)
  (< (+ (distance-to a) (distance-to b))
     (+ w (broadcast a (distance-to b)))))
```

Here distance from source fields `a` and `b` creates a Boolean field holding `true` only in those devices whose distance to `a` and `b` is less than "width" `w` greater than the shortest path. Such higher-level patterns can themselves be further modulated and combined, e.g. restricting a channel with `if` in order to circumvent an area considered to be an obstacle:

```
(def channel-with-obstacle (a b w obstacle)
  (if obstacle false (channel a b w)))
```

Expressing these in GPI-calculus ensures less fragility of the channel to device position: a near-identical naive channel program using = instead of < produces fragile channels that can disconnect or re-route due to minuscule perturbations. In GPI-calculus, however, this fragility is extinguished because = can never return *true*, only *false* and $\mathcal{B}$, rendering naive channel obviously unable to produce any sort of channel, fragile or otherwise.

1) Application Scenario: Wireless Sensor Network: Consider a wireless sensor network in which some devices must exchange a large amount of information, e.g., relaying video to a mobile monitoring station. The set of devices to relay to be identified with `channel-with-obstacle`, balancing limited spreading of information (e.g., to save battery energy) with replication along the transmission path (e.g., to increase reliability), and avoiding devices that do not wish to participate (e.g., due to low battery or faults). A `broadcast` restricted to this channel with `if` can then relay data with replication to prevent data loss, but much less resource consumption than unrestricted broadcast. Figure 6 shows an example simulated on a heterogeneous network: note how density affects only the precision of the channel's boundaries.

We confirm our GPI-calculus results using simulation with Alchemist [28] and Protelis [29]. We test `channel-with-obstacle` and the naive non-GPI-calculus variant on nine logarithmically scaled densities,

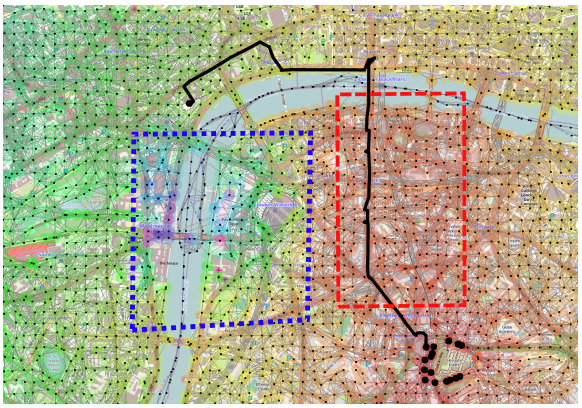
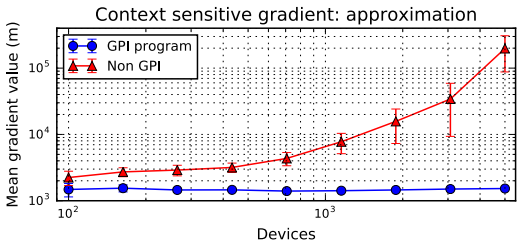
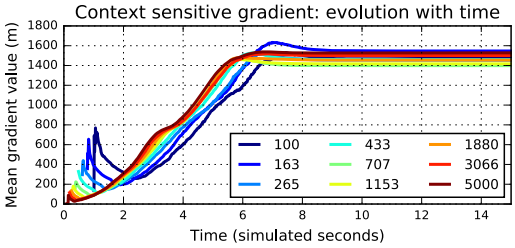


Fig. 8. Context-sensitive distance computation navigating the streets of London: warmer colored devices have a closer effective distance to the destination (black dots at bottom center). Dashed outlines are unfavourable (blue) and favourable (red) areas for travel. An example path is shown (black line), originating near Charing Cross (black dot in upper left).



(a) GPI vs. non-GPI convergence



(b) Convergence vs. time

Fig. 9. Simulation of urban traffic steering scenario confirms the analytical results for GPI-calculus, showing convergence with respect to both number of devices (a) and time (b). Number of devices shows mean and ± 1 standard deviation for both GPI (blue) and non-GPI (red) versions. Time graphs show mean of GPI version only, colored by number of devices shading from deep blue (100) to dark red (5000).

from 100 to 5000 devices, with ten runs per condition, distributing devices randomly. Devices are connected with a unit disc network, using a range of 15% of environment width at lowest density and reducing proportional to square root of density to ensure a consistent expected number of neighbors. Devices run unsynchronized but with the same clock speed, frequency rising inversely proportional to communication range to keep information speed consistent.

By the results in Section IV, it should be the case that for GPI-calculus device values will self-stabilize to a fixed set of values, and that as the number of devices increases, the values converged to will themselves converge as the network more closely approximates continuous space. We test this by measuring a key application property, the fraction of devices in the channel. Figure 7 shows the GPI-calculus program converges with respect to both time and number of devices, confirming our predictions. The naive channel, however, shows a low and decreasing fraction of participating devices: even floating point addition errors are enough to disturb the fragile equality relation.

B. Context-Sensitive Distance

The effective shortest path between a node and the source of a `GPI` is not necessarily the physically shortest path. Rather, effective distance may be influenced by other properties of the environment, either negatively (e.g., obstacles, congestion, pollution, tolls) or positively (e.g., safety, dedicated lanes, beauty). The `density` argument of `GPI` allows such factors to be taken into consideration as a multiplicative "stretching" of the base physical distance metric. Assuming there are penalising areas (`cons`) and favourable areas (`pros`), both expressed as scalar fields with values between 0 (least significant) and 1 (most significant), then one form of context sensitive distance is:

```
(def contextual-distance (source pros cons)
  (GPI source 0 (+ 1.1 (- cons pros))
    (+ 1.1 (- cons pros))))
```

Since `GPI` accumulates values using a path-integral, the context-sensitive stretching is guaranteed to be resilient to distribution changes. A naive alternate counting "pros" and "cons" visited rather than integrating creates a density-sensitive distance function whose value could be radically changed by changes in device location or network density.

1) Application Scenario: Urban Traffic Steering: Consider guiding pedestrian or vehicle traffic in a complex urban environment. Devices are deployed along and around the streets, some with environmental sensors (e.g, for crowding, traffic, pollution); other parameters are drawn from distributed or cloud databases (e.g., for events, attractions, comments on an area). From these, devices can compute contextual pro and con fields for people navigating through the city, reflecting perceived distance toward a location by taking path desirability into account. Figure 8 shows an example simulated in the center of London, in which context-sensitive distance chooses a more favored path over alternatives that are shorter but less favored.

We validate the predictions of GPI-calculus in this scenario using the same simulation environment as for the wireless sensor network scenario, except that devices are distributed on a street map of London and the property measured is the average contextual distance value. As expected, Figure 9 shows the GPI-calculus program converges with respect to both time and number of devices, confirming our predictions. The naive context-sensitive distance measure, however, does not stabilize but grows as the number of hops through modulated space increases.

VI. Contributions

We have presented a sub-language of field calculus containing only programs resilient against changes in the number and distribution of devices in a network. This is a step towards a more general framework for supporting open ecosystems of pervasive wireless devices, which need to provide safe and resilient services despite running a shifting set of interacting services from many unrelated software suppliers. If it is possible to implicitly ensure all programs are resilient and composable, then it will greatly reduce the cost of providing reliable services in such environments.

In future work, we aim to extend the breadth of the results in this paper. Most importantly, the theory we present does not cover mobile devices. In practice, however, the mechanisms used often perform well on mobile devices, so there appears to be good prospect for extension. Similarly, the theory currently directly addresses only the limit of approximation, but these properties tend to indicate that an algorithm also behaves well in lower density networks and before it has finished converging, so it should be possible to identify properties directly pertinent to lower density performance. Finally, `GPI` only addresses one of the key self-stabilizing patterns identified in [2], and a clear area for extension is to deal with additional "building block" algorithms, and complementarily to consider how static analysis, testing, and model-checking techniques can be used to eliminate program faults before runtime.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298 HyVar (Damiani), ICT COST Actions IC1402 ARVI and IC1201 BETTY (Damiani), Ateneo/CSP project RunVar (Damiani), and 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 authorised 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) 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, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, 2015.

[2] M. Viroli, J. Beal, F. Damiani, and D. Pianini, "Efficient engineering of complex self-organising systems by self-stabilising fields," in *IEEE Conf. on Self-Adaptive and Self-Organising Systems*, 2015.

[3] F. Damiani, M. Viroli, and J. Beal, "A type-sound calculus of computational fields," *Science of Computer Programming*, vol. 117, pp. 17–44, 2016.

[4] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: a higher-order calculus of computational fields," in *Proceedings of FORTE 2015*, June 2015, pp. 113–128.

[5] J. Beal and M. Viroli, "Building blocks for aggregate programming of self-organising applications," in *IEEE Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, 2014, pp. 8–13.

[6] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501.

[7] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The tota approach," *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.

[8] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *IEEE Conf. on Intelligent Robots and Sys.*, 2007, pp. 2794–2800.

[9] M. Viroli, D. Pianini, and J. Beal, "Linda in space-time: an adaptive coordination model for mobile ad-hoc environments," in *Proceedings of Coordination 2012*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7274, pp. 212–229.

[10] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, Cambridge, MA, USA, 2001.

[11] D. Coore, "Botanical computing: A developmental approach to generating inter connect topologies on an amorphous computer," Ph.D. dissertation, MIT, Cambridge, MA, USA, 1999.

[12] D. Yamins, "A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems," Ph.D. dissertation, Harvard, Cambridge, MA, USA, December 2007.

[13] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Dec. 2002.

[14] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *Int.l Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004, pp. 78–87.

[15] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computations in space and space in computations," in *Unconventional Programming Paradigms*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3566, pp. 137–152.

[16] C. Lasser, J. Massar, J. Miney, and L. Dayton, *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[17] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[18] F. Damiani and M. Viroli, "Type-based self-stabilisation for computational fields," *Logical Methods in Computer Science*, vol. 11, no. 4, 2015.

[19] J. Fernandez-Marquez, G. Marzo Serugendo, S. Montagna, M. Viroli, and J. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing*, vol. 12, no. 1, pp. 43–67, 2013.

[20] J. Beal, "Programming an amorphous computational medium," in *Unconventional Programming Paradigms*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3566, pp. 121–136.

[21] ——, "A basis set of operators for space-time computations," in *3rd Spatial Computing Workshop (SCW 2010)*, Sept 2010.

[22] J. Beal, M. Viroli, and F. Damiani, "Towards a unified model of spatial computing," in *7th Spatial Computing Workshop (SCW 2014)*, May 2014.

[23] J. Beal, K. Usbeck, and B. Benyo, "On the evaluation of space-time functions," *The Computer Journal*, vol. 56, no. 12, pp. 1500–1517, 2013, doi: 10.1093/comjnl/bxs099.

[24] T. Henzinger, "The theory of hybrid automata," in *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, Jul 1996, pp. 278–292.

[25] B. MacLennan, "Continuous spatial automata," University of Tennessee, Knoxville, Tech. Rep. Department of Computer Science Technical Report CS-90-121, November 1990.

[26] E. F. Taylor and J. A. Wheeler, *Spacetime Physics: Introduction to Special Relativity*, 2nd ed. W. H. Freeman & Company, 1992.

[27] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[28] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with Alchemist," *Journal of Simulation*, 2013.

[29] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *ACM Symposium on Applied Computing 2015*, April 2015, pp. 1846–1853.