IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

Self-adaptation to device distribution in the internet of things

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1649733 since 2017-10-16T22:20:58Z

Published version:

DOI:10.1145/3105758

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

02 May 2026

IRIS AperTO

UNIVERSITÀ
DEGLI STUDI
DI TORINO

This is the author's version of the contribution published as:

Beal, J., Viroli, M., Pianini, D., Damiani, F. Self-adaptation to device distribution in the internet of things. ACM Transactions on Autonomous and Adaptive Systems. Volume 12, Issue 3, September 2017.

doi: 10.1145/3105758

The publisher's version is available at:

https://dl.acm.org/citation.cfm?doid=3143529.3105758

When citing, please refer to the published version.

The final publication is available at

https://dl.acm.org/citation.cfm?doid=3143529.3105758

© ACM, 2017. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM Transactions on Autonomous and Adaptive Systems, { Volume 12, Issue 3, (September 2017)}
http://doi.acm.org/10.1145/3105758

Self-adaptation to Device Distribution in the Internet of Things

Jacob Beal, Raytheon BBN Technologies, USA
Mirko Viroli, University of Bologna, Italy
Danilo Pianini, University of Bologna, Italy
Ferruccio Damiani, University of Torino, Italy

A key problem when coordinating the behaviour of spatially-situated networks, like those typically found in the Internet of Things (IoT), is adaptation to changes impacting network topology, density, and heterogeneity. Computational goals for such systems, however, are often dependent on geometric properties of the continuous environment in which the devices are situated, 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 converges to a final state that approximates a predictable limit, based on the continuous environment, 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 [Beal et al. 2015; Viroli et al. 2015a] which identify a class of self-stabilizing programs. Finally, we confirm through simulation of IoT application scenarios that eventually consistent programs from this class can provide resilient behavior where programs that are only converging fail badly.

CCS Concepts: **•Theory of computation → Distributed algorithms; •Computing methodologies → Distributed computing methodologies;**

Additional Key Words and Phrases: Field calculus, self-organisation, self-stabilization, spatial computing, large-scale coordination

ACM Reference Format:
Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani, 20xx, Self-adaptation to Device Distribution in the Internet of Things *ACM Trans. Autonom. Adapt. Syst.* 0, 0, Article 00 (20XX), 31 pages.
DOI: 0000001.0000001

1. INTRODUCTION

The advent of the Internet of Things (IoT) is bringing us a dramatic increase of density of computational devices deployed in our cities, living and working environments. This kind of network, which we refer to as a (spatially-)*situated network*, poses novel engineering challenges, particularly in the area of distributed coordination: interactions encompass a large-scale system, and often need to be opportunistic, context-dependent, and based on physical proximity, which implies they must be adaptive and resilient to nearly continual faults and changes in the network. Such networks are also highly het-

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.
Author's addresses: J. Beal, Raytheon BBN Technologies, USA; M. Viroli and D. Pianini, University of Bologna, Italy; F. Damiani, University of Torino, Italy.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 20XX ACM. 1556-4665/20XX/-ART00 $15.00
DOI: 0000001.0000001

erogeneous 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. Prior work formally addressing resilience mostly considered the notion of self-stabilization [Dolev 2000], and its application to programming situated networks [Viroli and Damiani 2014; Viroli et al. 2015a; Damiani and Viroli 2015], guaranteeing that a program eventually reaches a correct final state independent of initial state, and hence of transient changes. However, this notion does not address the sensitivity of the final state to details of how devices are distributed in space and time.

In this paper we thus introduce a new property, *eventual consistency*, which guarantees that a computation executed by an IoT network not only eventually converges, but also that its values are a good approximation of executing the 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. We additionally identify sufficient conditions for eventual consistency, by considering the framework of aggregate programming [Beal et al. 2015], which factors distributed systems development into several layers: *field calculus* gives a universal computational model that maps between aggregate-level computations and local interactions between individual devices to implement those computations [Damiani et al. 2016; Damiani et al. 2015], systems of composable "building block" operators implemented with field calculus constructs provide resilience and scalability guarantees [Beal and Viroli 2014; Viroli et al. 2015a], and domain-general and domain-specific APIs built using building block operators provide a programatic interface for construction of complex networked services and applications [Beal and Viroli 2014; Viroli et al. 2015a]. We thus follow this framework to provide eventual consistency for spatially-situated networks by means a specific set of building blocks, whose arbitrary composition is proved to ensure eventual consistency of any application built with those building blocks.

Following a brief review of related work in Section 2, we provide a formal model of continuum computation and use this to define *eventual consistency* in Section 3. Section 4 identifies a subset of the field calculus that provides eventual consistency. Finally, Section 5 demonstrates the breadth of this sub-language and empirically confirms the value of eventual consistency in simulation of IoT application scenarios, and Section 6 summarizes the contributions of this paper and discusses future work. Note that this paper is an extended version of [Beal et al. 2016]: in this extended version we provide an in-depth discussion on eventual consistency, generalization to a much larger class of eventually consistent programs (including wide sets of functional operators), and a full proof of the eventual consistency of GPI-calculus, plus an extended empirical evaluation addressing networks with heterogeneous density as well as networks whose density changes over time.

2. RELATED WORK

A wide range of aggregate programming methods have been developed for engineering networks of devices physically-situated in space. A thorough review may be found in [Beal et al. 2013], which identifies four main approaches to aggregate programming. First, a number of "bottom-up" methods implement computational fields using

only the local view, including the Hood sensor network abstraction [Whitehouse et al. 2004], Butera's "paintable computing" hardware model [Butera 2002], TOTA [Mamei and Zambonelli 2009], the chemical models in [Viroli et al. 2015b], and Meld [Ashley-Rollman et al. 2007].

Complementary to these bottom-up methods are three families of "top-down" approaches, which specify tasks for aggregates and then rely on compilers or similar software to translate from aggregate specifications into a set of individual local actions that can implement the desired aggregate behavior. All of these tend to demonstrate significant resilience to changes in the distribution and density of devices. One of these families of approaches focuses on the creation of geometric and topological patterns, such as the geometric patterns of Origami Shape Language [Nagpal 2001], topological networks of Growing Point Language [Coore 1999], Yamins' universal patterns [Yamins 2007], or the self-healing geometries in [Clement and Nagpal 2003] and [Kondacs 2003].

A largely disjoint family of approaches focus instead on summarizing and streaming information over regions of space and time. Early examples of this approach are TinyDB [Madden et al. 2002] and Cougar [Yao and Gehrke 2002], which enable querying a sensor network by expressing a high-level declarative query that is then automatically compiled into a set of low-level sensor activities of data collection and aggregation; these have been followed by other methods offering more complex functionalities (e.g., [Newton and Welsh 2004; Curino et al. 2005]).

Generalizing on both of these classes are a collection of general purpose space-time computing models. These include explicitly spatial parallel computing models, most notably StarLisp [Lasser et al. 1988] and systolic computing (e.g., [Engstrom and Cappello 1989; Raimbault and Lavenier 1993]), which use parallel shifting of data on a structured network. The MGS language [Giavitto et al. 2002; Giavitto et al. 2005] takes a notable and different approach, evolving the shape of the manifolds on which it executes. General purpose space-time computing models, and most particularly field calculus [Damiani et al. 2016; Damiani et al. 2015] have been the basis for a layered approach to building distributed adaptive systems as presented in our previous work [Beal et al. 2015; Viroli et al. 2015a].

More generally, identifying robust (e.g., self-adapting or self-stabilizing) algorithms for distributed systems is a long investigated problem [Dolev 2000], which is part of the general challenge of devising sound techniques for engineering self-organising applications. Several techniques can be used, spanning game theory (e.g., [Yen et al. 2016]), SMT-based automatic synthesis (e.g., [Faghih and Bonakdarpour 2015]), and order statistics (e.g., [Faghih and Bonakdarpour 2015]).

Collective adaptive behaviour involving groups of computational entities spread in an (physical or virtual) environment has also been the subject of deep study in the multi-agent systems literature, where a huge variety of techniques and approaches are used: coordination artifacts of various form [Omicini et al. 2008], protocols [Kalia and Singh 2015]), social/organisational norms [Artikis et al. 2009], commitments [Mallya and Singh 2007], swarm intelligence [Parunak et al. 2005], and teamwork [Lesser et al. 2004; Taylor et al. 2011]). The work in [Viroli et al. 2015c] defines a notion of aggregate plan as a space-time structure of "actions": since it is based on aggregate programming, it is the approach where the results of this paper are most readily applicable.

In this paper we are concerned in finding large sufficient conditions for resiliency, expressed in terms of a whole language of resilient programs that can be constructively used to build systems. To the best of our knowledge, however, the only works aiming at a proof of resilience for an entire class of computational field algorithms are [Damiani and Viroli 2015] and [Viroli et al. 2015a], which address self-stabilization. In particular, these works consider deterministic self-stabilization, in which the system

Symbol	Definition
$\overline{x}$	Closure of a set; sequence of a token
M	A Riemannian space-time manifold
m	A point ("event") in a manifold M
c	Bound on the speed of information propagation
$T^+(m)$	Time-like future: events reachable from m slower than c
d	Computational device: a time-like curve in a manifold M
S_M	A space-like cross-section of a manifold M
D	Discrete subset of a manifold, e.g., events in execution of an algorithm on physical devices.
$\mathbb{V}$	Set of all possible data values
f	Field: a function $f : M \to \mathbb{V}$ assigning values to all points in the manifold M.
$\mathbb{F}_{\mathbb{V}}$	Space of all fields with range contained in $\mathbb{V}$
e	"Evaluation environment" input field
C	Computation: a higher-order function mapping fields to fields: $C : \mathbb{F}_{\mathbb{V}} \to \mathbb{F}_{\mathbb{V}}$

Fig. 1. Key symbols used in the definition of eventual consistency.

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 [Fernandez-Marquez et al. 2013], 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.

As a starting point for addressing eventual consistency, then, a model for computation in space is needed. There are a number of continuum computation models that might be considered, including hybrid automata [Henzinger 1996] and continuous spatial automata [MacLennan 1990]. In this paper, we follow the direction proposed in [Abelson et al. 1999] and, more particularly, its realization in [Beal 2005; 2010], in which computation is seen as a field mapping a continuous region of space-time (namely, a *manifold*) to data values. This approach, we shall see in detail later, fits naturally with the field calculus, as well as with the notion of computation self-adapting to device distribution changes.

3. EVENTUAL CONSISTENCY

Self-stabilization is a well-established theoretical property of distributed algorithms [Dolev 2000]: 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 (for technicalities of the particular definition of self-stabilization referred to by this paper, see [Viroli et al. 2015a]). We now introduce a new and closely related property, *eventual consistency*, that ties 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 IoT scenarios well, since many computations are more concerned with the environment in which devices are deployed rather than the particulars of individual devices, and can hence be naturally described in geometric terms, e.g., distances, regions, information flow.

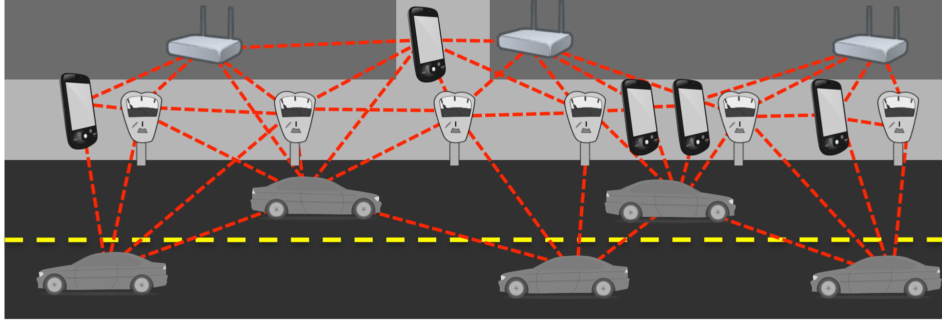


Fig. 2. An IoT example in a street environment; the walkable spaces of the street form a manifold.

We will develop the concept of eventual consistency incrementally. First, we review how situated networks can be viewed as an approximation of a continuous environment, which then forms a basis for both discrete and continuous models of space-time computation. 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. For additional assistance to the reader, Figure 1 provides a table of key symbols.

3.1. 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 [Beal 2005] and [Beal 2010], a network can be viewed as a discrete approximation of the continuous physical space and programmed accordingly.

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 (e.g., the shape of a city's streets or a building's interior). 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 2) 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 may be modeled as a bound c on the speed at which information can propagate. A standard set of concepts and terminology from relativistic physics (e.g., [Taylor and Wheeler 1992]) may then be borrowed to describe the space-time relations of a manifold. 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 3):

— 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 $T^+(m)$, 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.

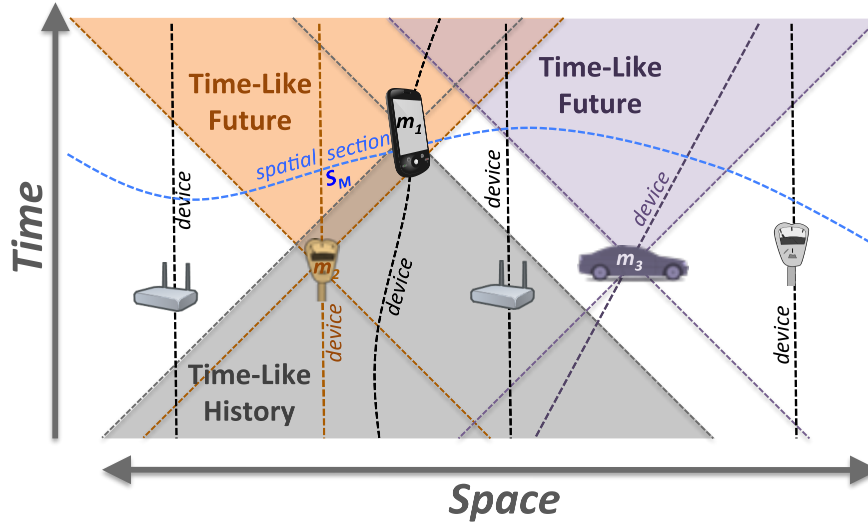


Fig. 3. Example of continuous space-time relations between six devices distributed along a street: wireless access points and meters are stationary, while the car moves steadily and the phone stops and starts twice.

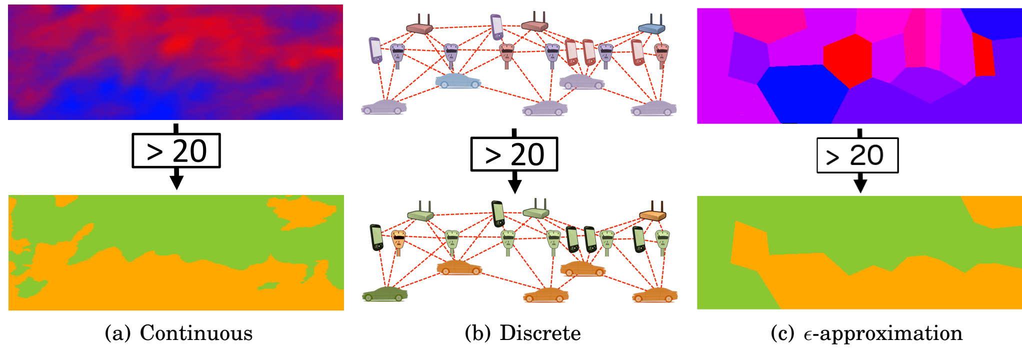
Finally, we also define a *spatial section* S_M, a distributed snapshot of state without a global notion of time: formally, a spatial section is any set S_M in which no event is in any other's time-like future or history, namely, such that $T^+(S_M) \cup T^-(S_M) = M - S_M$.

In this model, a *device* d is represented with a one-dimensional time-like curve in the manifold, analogous to the physics notion of a world-line. Devices thus can have a history in time, but only exist at one point in space at any given time. The mathematical analysis in this paper makes the simplifying assumptions that the manifold is finite in diameter and devices do not move, which better fits the general framework of self-stabilization and convergence (in practice the results we provide typically also apply well to devices moving much more slowly than c).

Figure 3 depicts an IoT computing example of these concepts in terms of several devices distributed in 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 (m_1) is in the time-like future of the marked event on the orange meter's history (m_2), 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 (m_3), meaning it only has access to older information about the car. Finally, the blue line marks one of many possible spatial section "snapshots" (S_M) that can separate all of space-time into a "strict future" and "strict past."

3.2. Computations Across Space and Time

Standard event-based models of distributed computation (e.g., [Lynch 1996]) 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 [Beal 2010; Beal et al. 2014; Beal et al. 2013]:



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

Fig. 4. On the situated network in a street environment in Figure 2: a continuous computation of a temperature threshold (b) is approximated by the discrete network of devices (c), producing the ϵ-approximation shown in (d).

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 $\mathbb{V}$, where $\mathbb{V}$ is the set of all possible data values.*

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 $D \subset M$. Note that defining discrete execution as a manifold subspace (rather than an abstract graph) preserves the geometric relationship of network to environment.

For example, consider a situated network in a street environment such as is shown in Figure 2. Two examples of continuous computational fields are shown in Figure 4(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 4(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 4(a) and 4(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 Program) *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. A space-time program is 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.

This is much like the definition of a computation, except that the domains of the fields may differ. Note also that this includes recursive composition, e.g., via lambda calculus, so programs are potentially universal, per [Beal et al. 2014]. Some further notes on technicalities: first, any well-defined composition of operators is itself an operator; second, complete programs have no inputs except the environment. Note, however, that any composition of operators that requires inputs (e.g., function definitions) may be transformed into an equivalent program with no inputs, in which these values are instead supplied by the environment and a special "no value" value is adjoined for events in the domain of the environment but not the input.

For example, the space-time computation depicted in Figure 4 can 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.

3.3. 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.

Note that we consider only causal computations, i.e., those whose results only depend on information from the past and present. This is not a significant restriction since acausal computations, while well-defined, cannot generally be implemented because they use information from the future, and so are not generally of interest when considering real-world systems. To be precise: a computation C is causal when for any two fields f and f' with the same domain M, it is the case that for every $m \in M$, if f and f' are equal on the closure of their time-like history $\overline{T^-(m)}$, then the outputs $C(f)$ and $C(f')$ are also equal at m. See [Beal 2010] for more information on the subject of causal computations.

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 4(c), which illustrates an ϵ-approximation of the fields of the discrete computation in Figure 4(b) on the manifold of the environment illustrated in Figure 2. Notice that this is a coarse approximation of the continuous computation in Figure 4(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 over manifolds M_i, as $\epsilon_i \to 0$, if both the following hold:*

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

$$\lim_{i \to \infty} \int_{M \cap M_i} \mu_{\mathbb{V}}(f, f_i) \;=\; 0$$

where $\mu_{\mathbb{V}}$ is a metric function over $\mathbb{V}$.

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. 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.

Two important technical notes about field approximation: first, note that the integral in the second function is a Lebesgue integral (see e.g., [Kestelman 1960]), since the more familiar Riemann integral is not well-suited for use on manifolds and is ill-defined on many discontinuous fields; for those unfamiliar with Lebesgue integrals, however, the intuitions of Riemann integration should generally serve. Note also that any metric function may be used for $\mu_{\mathbb{V}}$, since the properties of metric functions guarantee that the limit of the function can only go to zero when field values become infinitesimally close or identical almost everywhere.

With this definition in hand, we can define a *consistent program* as one where field approximation of inputs implies field approximation of outputs:

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

For most programs involving communication, however, consistency cannot be guaranteed without further specification of the communication model. This is because in many communication models, even minor shifts in device position can lead to information moving at significantly different speeds. Regardless of the specifics of communication, however, a program that converges to a steady state may be consistent after it converges:

Definition 7 (Eventually Consistent Program) *Consider a causal program P evaluated on domain M. Program 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 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 4 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) 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 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.

For contrast, here is the a definition of self-stabilization, as adapted from [Viroli et al. 2015a] into this model:

Definition 8 (Self-Stabilizing Program) *Consider a causal program* `P` *evaluated on domain M. Program* `P` *is* self-stabilizing *if there is a function $R : \mathbb{F}_{\mathbb{V}} \rightarrow \mathbb{F}_{\mathbb{V}}$ from fields on spatial sections to fields on spatial sections, such that for any evaluation environment e with a spatial section S_M where 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* `P` *is equal to $R(e|S_M)$ on every spatial section in the time-like future $T^+(S'_M)$—where $e|S_M$ means the field e with its domain restricted to S_M.*

In other words: if the inputs ever converge, then the outputs eventually converge to values fixed by the final set of input values. This is different from eventual consistency in two ways: first, self-stabilization is "memoryless," considering only the final set of input values, and second, self-stabilization implies nothing about consistency across network structures.

In particular, note that eventual consistency does not necessarily imply self-stabilization, as the value to which an eventually consistent system converges is still allowed to depend on the history of values in its environment. The example just given of a gossip algorithm that computes whether any location has ever seen a high temperature is one such program that is eventually consistent but not self-stabilizing. The approach that we present in the next section for obtaining eventual consistency, however, will ensure self-stabilization as well.

4. 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. Following a brief review of field calculus (the basis of our approach), we analyze how field calculus 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 of field calculus that contains only eventually consistent programs, called GPI-calculus.

4.1. Field Calculus

Field calculus [Damiani et al. 2016; Damiani et al. 2015] is a minimal universal language, in which every expression specifies a space-time program, as defined in Section 3.2. 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 [Beal et al. 2014], which means that it is a good framework for investigating eventual consistency.

```
e ::= x | l | (b e1 ... en) | (f e1 ... en)      ;; expression
      | (if e e e)                                ;; restriction
      | (rep x w e) | (nbr e)                     ;; space-time ops
w ::= x | l                                       ;; variable or value
F ::= (def f(x1 ... xn) e)                        ;; function
P ::= F1 ... Fn e                                 ;; program
```

(a) Syntax of Field Calculus

```
e ::= x | l | (b e1 ... en) | (f e1 ... en)      ;; expression
      | (if e e e)                                ;; restriction
      | (GPI e e e e)                             ;; space-time op
F ::= (def f(x1 ... xn) e)                        ;; function
P ::= F1 ... Fn e                                 ;; program
-------------------------------------------------------------
Restricted values for l and b:
l ::= 𝔹 | ℝ | ℬ                                   ;; Literals
b ::= cf | df | s | d | sense                     ;; local operators
```

(b) Restriction to GPI-calculus sub-language

Fig. 5. Field calculus [Damiani et al. 2016] is a minimal computational calculus that does not ensure eventual consistency. GPI-calculus is a restriction to a sub-language of eventually consistent (and self-stabilizing) programs.

Field calculus programs are specified using the syntax in Figure 5(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 $(\mathtt{b}\ \mathtt{e}_1 \ldots \mathtt{e}_n)$ determines the value of its output field at event m only from the values of the environment e and input fields $\mathtt{e}_1, \mathtt{e}_2, \ldots$ 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 $(\mathtt{def}\ \mathtt{f}(\mathtt{x}_1 \ldots \mathtt{x}_n)\ \mathtt{e})$ and called with expressions of the form $(\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n)$.
— *Time evolution:* Program state is initialized and changed over time by a "repeat" construct `(rep x w e)`, initializing `x` to a value `w` (supplied either by a variable or by a literal) and updating (non-synchronously) by computing `e` against its prior value.
— *Neighborhood values:* At each event, expression `(nbr 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), and to infinity if there are no neighbors.
— *Domain restriction*: $(\mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2)$ computes expressions in subspaces, preventing interference between the two sub-computations: $\mathtt{e}_1$ is computed where Boolean $\mathtt{e}_0$ is true, $\mathtt{e}_2$ 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 3.3 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
```

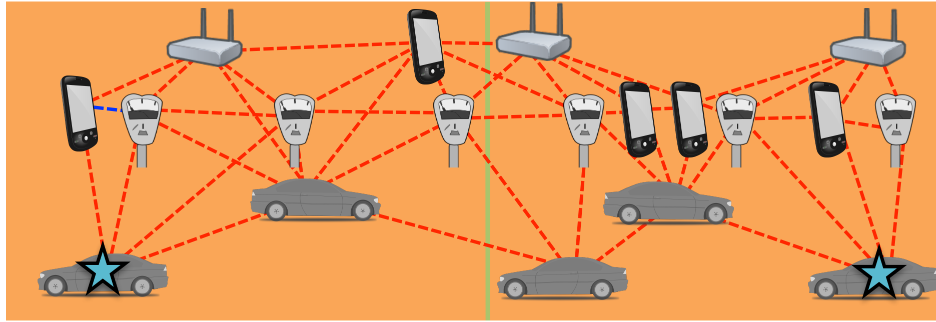


Fig. 6. Finding a bisector is fragile because it is sensitive to device positions. For example, the set of bisecting locations (green line) for two cars (blue stars) might or might not actually include any devices.

```
    false
    (or ever (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 and *false* if all neighbors hold *false* or if there are no neighbors.

Because field calculus is universal, it can express any program, including non-resilient programs. A sub-language was identified in [Viroli et al. 2015a], 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 which cover a large number of self-stabilizing algorithms. The consistent calculus we identify below further restricts this self-stabilizing sub-language.

4.2. 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 programs that implicitly rely on the time between events, but the self-stabilizing sublanguage in [Viroli et al. 2015a] 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 and state constructs 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 6). 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). To handle such problems without giving up either useful discrete constructs or the connection to continuous environments, we must develop some means of dealing with the problems posed by boundaries.

4.3. Restriction of Field Calculus: GPI-calculus

We now further restrict field calculus to a sub-language of eventually consistent programs, which we call GPI-calculus, whose syntax is shown in Figure 5(b). This sub-language is also self-stabilizing, as it is also a restriction of the self-stabilizing sub-language of field calculus presented in [Viroli et al. 2015a]. 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 literal data values are restricted to Booleans $\mathbb{B}$ (the prototypical discrete value space) and real numbers $\mathbb{R}$ (the prototypical approximable continuous value space), plus a unique value $\mathcal{B}$ denoting a possibly problematic boundary between value regions.[1]

[1]Note that a wide range of other types of literals, including integers, tuples, and lists, can be generated from these literals and the built-in operators defined below. Note also that the issues of fragility would not be solved if we were to replace $\mathbb{R}$ with more computationally tractable alternatives like rationals or finite-precision floating point numbers.

— Built-in operators are restricted to the classes `cf` (continuous), `df` (discrete), `s` (selection), `d` (discretization), and `sense` (sensing)—all described in detail below.
— `if` allows $\mathcal{B}$ as an additional value for its first input other than Boolean true and and false; for those points mapping to $\mathcal{B}$, the output also maps to $\mathcal{B}$. Note that this does not actually change semantics from field calculus, as it can be implemented via syntactic sugar on two nested field calculus `if` statements.
— State and communication are only available indirectly through a new operator, `GPI` (described below), which is a restriction of the spreading pattern in [Viroli et al. 2015a].
— Recursion in any form is prohibited by the simple expedient of having the body of function definition F_i in the sequence of function definitions prohibited from referencing any function $F_{j \geq i}$.

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:

— `cf` is any strictly continuous mathematical function, extended to have output $\mathcal{B}$ if any input is $\mathcal{B}$. Examples include addition, multiplication, logarithm, and sine, as well as construction of tuples and extraction of tuple values from a fixed index, which can be used to implement data structures. Some simple functions are excluded, however, such as division, which is discontinuous when the denominator is zero.
— `df` is any discrete mathematical function, i.e., any function that takes only integers or Booleans for inputs and returns integers or Booleans as output, extended to have output $\mathcal{B}$ if any input is $\mathcal{B}$. Examples include integer arithmetic and Boolean logic operations.
— `s` is any "selection" function that takes an integer or Boolean as its first argument and uses its value to select between the values of other inputs, returning $\mathcal{B}$ when the first argument is $\mathcal{B}$. An example of `s` is the piecewise multiplexer function `mux`: where its first input is *true*, it returns the second input; where it is *false*, it returns the third input; where it is $\mathcal{B}$, it returns $\mathcal{B}$.
— `d` is any single-input "discretization" function that maps non-intersecting open intervals of the real numbers to constant integers or Boolean values (each open interval may map to a different constant), and maps every value not included in those open sets to $\mathcal{B}$. An example of `d` is the sign function, which maps all positive numbers to *true*, all negative numbers to *false*, and zero and $\mathcal{B}$ to $\mathcal{B}$ (note that this sign function can be composed with subtraction to create comparators). Another example is rounding, with the "halves" ($1/2$, $3/2$, $5/2$, etc.) mapping to $\mathcal{B}$, which can turn real numbers into integers.
— (`sense` k) returns the kth value in the environment state (assumed to be a tuple or $\mathcal{B}$), where k is the positive integer literal given as its input, or else $\mathcal{B}$ for any point where the environment state is $\mathcal{B}$.

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 [Viroli et al. 2015a]:

```
(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 and those devices are not all members of the same ray centered on the current device. In other words, if there is more than one shortest path from the source to a device, that device must be on a boundary in the space (where the integral may be discontinuous) and is marked accordingly.

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), and all points whose values might be ambiguous (due to the existence of 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 a neighbor on the (sole) minimal path to the source.

Importantly, just as with G in [Viroli et al. 2015a], 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 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 $e^{-1}(\mathbb{V} - \mathcal{B})$.*

The full proof of this theorem is given in Appendix A. In sketch form: we first consider any set of operators that are eventually consistent and are eventually continuity-preserving, in the sense that there is always a spatial section S_M such that if 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 their output f_o is also continuous on $f_o^{-1}(\mathbb{V} - \mathcal{B}) \cap T^+(S'_M)$. All finite compositions of such operators can be shown by induction to have the same properties of eventual consistency and eventual continuity-preservation. Each operator in GPI-calculus can then be shown to be at least eventually consistent and eventually continuity-preserving, in a lengthy but not particularly complicated 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.

5. VALIDATION AND APPLICATIONS

We now validate the predictions of eventual consistency and demonstrate some of the breadth of IoT 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 [Viroli et al. 2015a], 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 have a twofold goal:

(1) comparing GPI-calculus algorithms with similar ones that lack eventual consistency, in order to demonstrate the perils of fragility and our ability to overcome these problems, and
(2) demonstrating the anticipated resilience of GPI-calculus to differences in devices density over both space and time.

Together, these experiments 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 sorts of difficulties that can often arise if eventual consistency is left to programmers rather than being guaranteed by the framework.

5.1. 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 device 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 programs in GPI-calculus ensures less fragility of the channel to device position: a near-identical naive program using `=` instead of `<` produces frag-

ile 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.

5.1.1. 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 is identified using `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.

We confirm our GPI-calculus results using simulation with Alchemist [Pianini et al. 2013] and Protelis [Pianini et al. 2015]. We first compare `channel-with-obstacle` and the naive non-GPI-calculus variant on nine logarithmically scaled densities, 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.

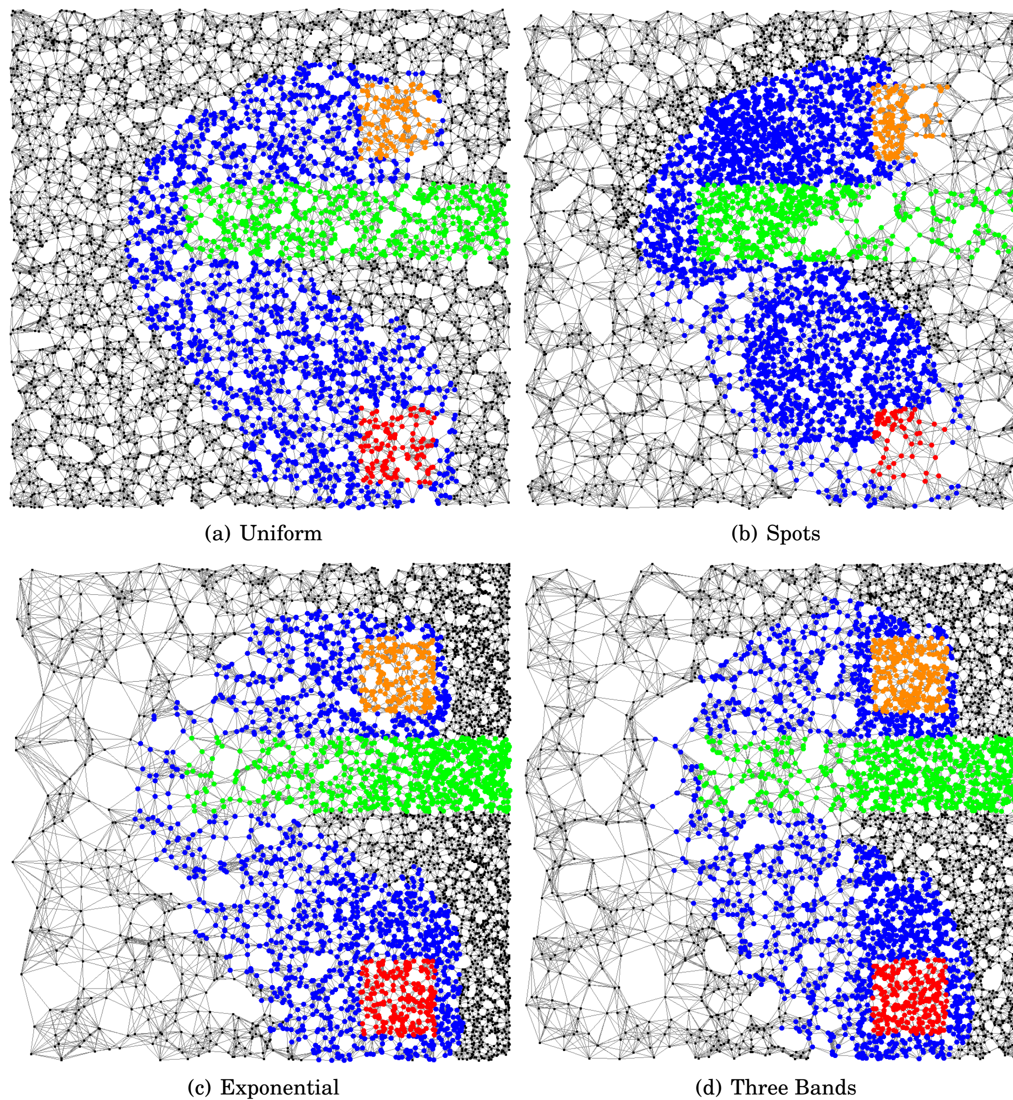
Our second set of tests then investigates how the GPI-calculus version of the program reacts to heterogeneity in space and time. To evaluate space heterogeneity, we run `channel-with-obstacle`, ranging the overall number of devices within the same range of values of the previous experiment, using four different network configurations:

(1) devices spread uniformly randomly through the space;
(2) same as above, but with three randomly chosen high-density regions;
(3) devices spread uniformly randomly along the Y-axis and exponentially randomly ($\lambda = 0.25$) along the X-axis;
(4) devices spread uniformly randomly in three vertical "bands" whose average density was determined by the same $\lambda = 0.25$ exponential random function.

Figure 7 shows snapshots of `channel-with-obstacle` simulations executed on such configurations: note how changes in density affect only the precision of the channel's boundaries.

For time heterogeneity, we run our experiments with 5000 devices scattered uniformly randomly in the space and perturb the overall number of active devices between 5000 and 1000 active devices using three different drivers: a square wave, a sine wave, and a triangle wave. Due to the sometimes extreme differences in density, we used a different connection rule for these experiments in order to prevent network segmentation: we dynamically linked device d_1 and d_2 if both were active and either one belonged to the set of the ten devices closest to the other. In this setup, we do not attempt to keep information speed consistent, and simply run all devices unsynchronized at 1 Hz frequency.

By the results in Section 4, 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 8 shows that, as expected, the GPI-calculus program converges with respect to both time and number of devices, confirming our



(a) Uniform (b) Spots

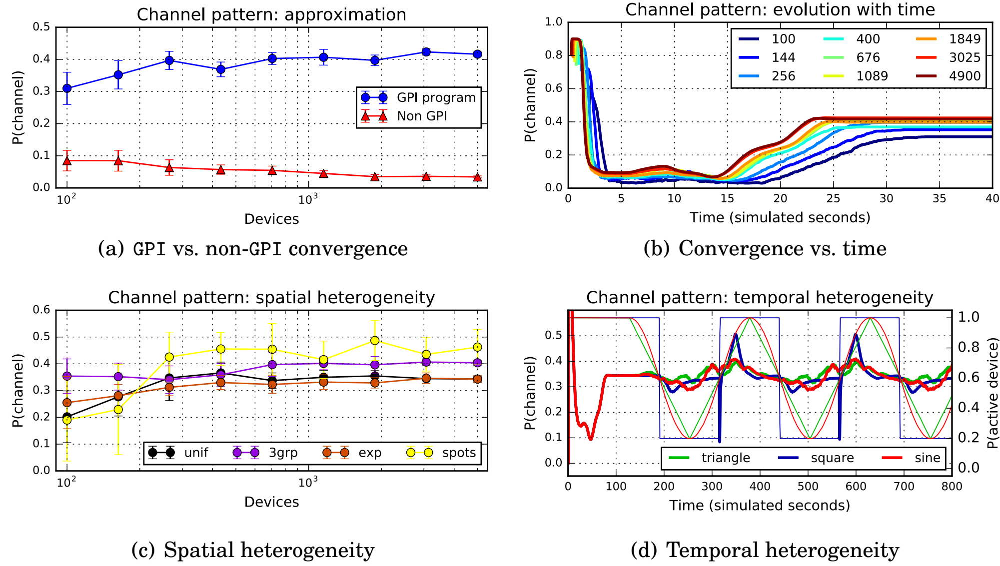
(c) Exponential (d) Three Bands

Fig. 7. Simulation snapshots of `channel-with-obstacle` exemplifying the four different device distributions used for the spatial heterogeneity tests. The obstacle is green, regions `a` and `b` are red and orange, and the computed channel is blue.

predictions. The GPI-calculus version of the algorithm also proves to be significantly resilient to heterogeneity in both spatial and temporal device distribution. The naive channel, however, shows a low and decreasing fraction of participating devices: even the minuscule imprecisions of floating point addition are enough to disturb the fragile equality relation.

5.2. Context-Sensitive Distance

For a second application example, we begin by considering the fact that 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



(a) GPI vs. non-GPI convergence

(b) Convergence vs. time

(c) Spatial heterogeneity

(d) Temporal heterogeneity

Fig. 8. Simulation of a wireless sensor network scenario confirms our analytical results for GPI-calculus, showing convergence with respect to both number of devices (a) and time (b). Different patterns of heterogeneity in space (c) and time (d) have a moderate impact on the result, but the differences decrease at higher densities and over periods of relative stability. Number of devices and spatial heterogeneity show mean and ± 1 standard deviation for, respectively, the GPI (blue) and non-GPI (red) versions, and for the GPI version in the different spatial configurations. Time graphs show mean of GPI version only; a comparison with non-GPI that includes ± 1 standard deviation is available in Appendix B. In (b), colors indicate different number of devices, shading from deep blue (100) to dark red (5000). In (d), different colors are associated to different perturbing drivers (thin lines, measured as probability for a node to be enabled). Results are the average of 10 simulation runs.

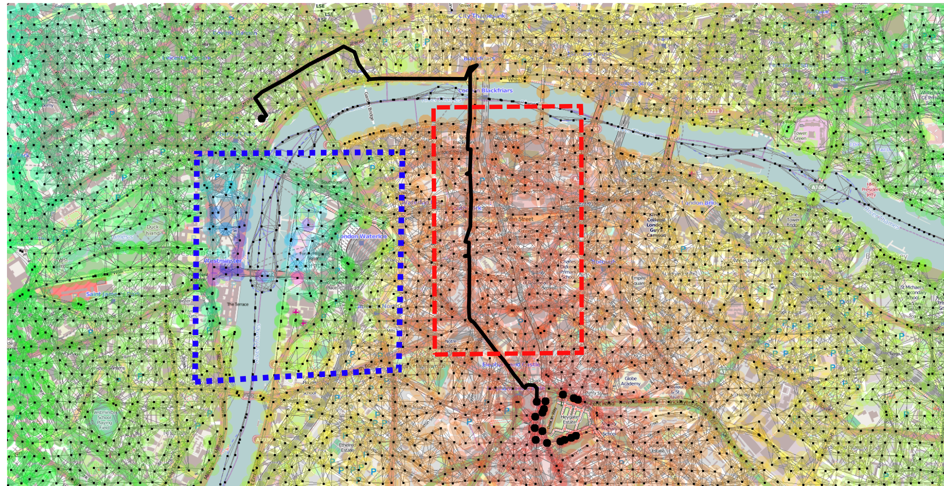
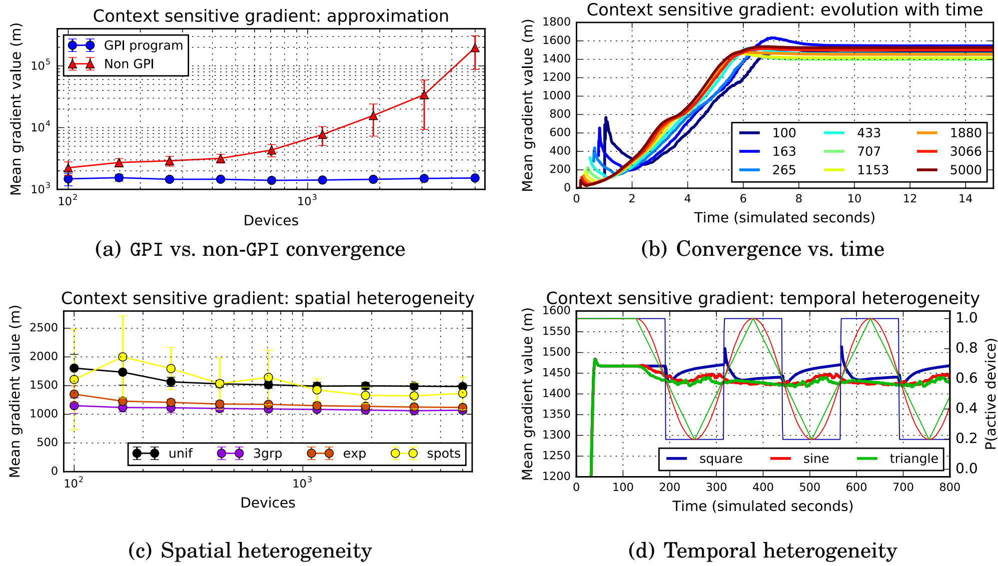


Fig. 9. Context-sensitive distance computation navigating the streets of London: warmer colored devices have a closer effective distance to the destination (black dots at bottom center). Dashed outlines are unfavourable (blue) and favourable (red) areas for travel. An example path is shown (black line), originating near Charing Cross (black dot in upper left).



(a) GPI vs. non-GPI convergence

(b) Convergence vs. time

(c) Spatial heterogeneity

(d) Temporal heterogeneity

Fig. 10. 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); and resilience to spatial (c) and temporal (d) heterogeneity. Number of devices and spatial heterogeneity show mean and ± 1 standard deviation for, respectively, the `GPI` (blue) and non-`GPI` (red) versions, and for the `GPI` version in the different spatial configurations. Time graphs show mean of `GPI` version only; a comparison with non-`GPI` that includes ± 1 standard deviation is available in Appendix B. In (b), colors indicate different number of devices, shading from deep blue (100) to dark red (5000). In (d), different colors are associated to different perturbing drivers (thin lines, measured as probability for a node to be enabled). Results are the average of 10 simulation runs.

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.

5.2.1. 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 9 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 10 shows convergence with respect to both time and number of devices for the GPI-calculus program. Similarly to the previous scenario, the `GPI` version is also resilient to heterogeneity in space and time, again confirming our predictions. The naive context-sensitive distance measure, on the other hand, does not stabilize but instead grows as the number of hops through modulated space increases.

6. CONTRIBUTIONS

We have presented a sub-language of field calculus containing only programs resilient against changes in the distribution (density, heterogeneity, topology) of devices in a network. This is a step towards a more general framework for supporting open ecosystems of pervasive wireless devices for the IoT, 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 (as illustrated by our heterogeneity results), so it should be possible to identify properties directly pertinent to lower density performance. Also, as we here focus only on eventuality of consistency, future works will also devoted to study performance issues to guarantee prompt reactiveness to classes of changes, either by creating alternative, optimized versions of building blocks [Viroli et al. 2015a], or by meta-techniques for optimization along the lines of [Pianini et al. 2016]. Finally, `GPI` only addresses one of the three key self-stabilizing patterns identified in [Viroli et al. 2015a], 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.

APPENDIX

A. EVENTUAL CONSISTENCY OF GPI-CALCULUS

We prove eventual consistency of GPI-calculus in three stages: First, we prove that any finite composition of eventually consistent and continuity-preserving operators is also eventually consistent and continuity-preserving. We then show that each operator in GPI-calculus is individually at least eventually consistent and continuity-preserving. Finally, we show that all GPI-calculus programs are finite compositions of operators, which implies that all such programs are eventually consistent.

Let us begin with a formal definition of what we need from operators in terms of preservation of continuity:

Definition 9 (Continuity-Preserving) *A space-time operator* $o : e \times f_i^k \rightarrow f_o$ *is* continuity-preserving *if it is the case that when environment* e *and inputs* f_i *are con-*

tinuous on $e^{-1}(\mathbb{V} - \mathcal{B})$ *and* $f_i^{-1}(\mathbb{V} - \mathcal{B})$*, then it is the case that the output* f_o *is also continuous on* $f_o^{-1}(\mathbb{V} - \mathcal{B})$*.*

Intuitively, what this means is that the output is a collection of continuous regions, "stitched together" on by regions of the boundary value $\mathcal{B}$. Note that we can turn any operator into a continuity-preserving operator simply by mapping every potential area of discontinuity in the output to the boundary value $\mathcal{B}$.

We may not, however, be able to guarantee continuity preservation immediately, but only after some time for convergence:

Definition 10 (Eventually Continuity-Preserving) *A space-time operator* $o : e \times f_i^k \rightarrow f_o$ *is* eventually continuity-preserving *if there is always a spatial section* S_M *such that* o *is continuity-preserving on* $T^+(S_M)$*.*

In addition to these two, we note that, by the definitions of space-time operators and programs in Section 3, consistency and eventually consistency (Definitions 6 and 7) can be extended to apply to operators simply by extending the approximability condition for consistency to apply to inputs as well as the environment.

With these definitions in hand, we now begin our proof of the eventual consistency of GPI-calculus by showing that sense operators, which merely access the environment, are trivially both consistent and continuity-preserving:

Lemma 2 *Built-in operator* `sense` *is consistent and continuity-preserving.*

PROOF. The environment is assumed to be a field mapping to a tuple of values at each point or to $\mathcal{B}$, and `sense` outputs a field created by indexing into said tuples by a literal integer, or returning $\mathcal{B}$ for points that are $\mathcal{B}$ in the environment. By the definition of consistency (Definition 6), we are only concerned with cases in which the environment is approximable, and a field of tuples cannot be approximable unless a field of each of the elements is also approximable. Likewise, a field of tuples cannot be continuous unless a field of each of the elements is also continuous. Thus, copying an element out of the environment for all non-$\mathcal{B}$ values must result in an output field that is also consistent and continuous on all non-$\mathcal{B}$ values. □

We next show that eventual consistency and eventual continuity are both preserved by composition:

Lemma 3 *Any program defined as a finite composition of eventually consistent and eventually continuity-preserving operators is also eventually consistent and eventually continuity-preserving.*

PROOF. Consider a well-defined program comprised of a finite sequence of operator instances, meaning that every operator instance input is defined as an output from some other operator instance and there are no cycles. Because the sequence is finite and the program is well-defined, it must be the case that the operator instances can be ordered, such that the ith operator instance depends only on the environment and the output fields of prior operators in the sequence.

For the ith operator instance in this sequence, it is possible to construct an equivalent program comprising only that operator instance, an environment e_i containing its inputs, and `sense` operator instances mapping the environment values to its inputs. By Lemma 2, the `sense` operators are consistent and continuity-preserving, so the ith operator instance has the input conditions for continuity preservation satisfied.

Thus, we also have that this miniature program is eventually consistent and eventually continuity-preserving.

The output of the ith operator instance can then be added to the environment for the $i+1$st operator instance, ensuring that if the preconditions are satisfied for the ith operator instance, they will be satisfied for the $i+1$st operator instance as well. The base case of $i=1$, of course, is true by assumption, since each operator is individually eventually consistent and eventually continuity-preserving. Thus any finite composition of operators is eventually consistent and eventually continuity-preserving if these properties hold for all of the individual operators. □

Note that if the program were not guaranteed finite evaluation, then this result would not hold: the eventual consistency of the ith instance evaluated would still hold, but no i would be high enough to cover the entire sequence, and it would be possible to construct sequences that do not converge.

We now move on to proving that all of the rest of the operators in GPI-calculus are sufficiently consistent and continuity-preserving, beginning with the trivial case of literals:

Lemma 4 *Literals* `l` *are consistent and continuity-preserving.*

PROOF. Trivially true, since the output field of a literal `l` is equal to `l` at every point in its domain, and constant functions are continuous. □

Lemma 5 *Any built-in operator* `cf` *is consistent and continuity-preserving.*

PROOF. Recall that `cf` is defined to be any strictly continuous mathematical function, extended to have output $\mathcal{B}$ if any input is $\mathcal{B}$. The output of any `cf` operator at any event m is affected only by the values of its inputs at m. `cf` is also continuous by definition, and (by the semantics of field calculus) requires all inputs to have the same domain. Thus, since the composition of continuous functions is continuous, when each input f_i is continuous on $f_i^{-1}(\mathbb{V}-\mathcal{B})$, it must be the case that the output f_o is continuous on $\bigcap_i f_i^{-1}(\mathbb{V}-\mathcal{B})$. The complementary space, $\bigcup_i f_i^{-1}(\mathcal{B})$ covers only points where at least one input has value $\mathcal{B}$, and thus by the definition of `cf`, f_o maps all events in this space to $\mathcal{B}$.

Only consistency remains to be shown. All ϵ-approximation sequences must approximate f_o on subspace $f_o^{-1}(\mathbb{V}-\mathcal{B})$ because it is continuous on this space. For the complementary space $\bigcup_i f_i^{-1}(\mathcal{B})$, f_o must also always be approximable because it holds the constant value $\mathcal{B}$ on a finite union of subspaces of approximable fields. □

Lemma 6 *Any built-in operator* `df` *is consistent and continuity-preserving.*

PROOF. Recall that `df` is defined to be any mathematical function from discrete inputs to discrete outputs, extended to have output $\mathcal{B}$ if any input is $\mathcal{B}$. By the semantics of field calculus, its inputs are all required to have the same domain. By the precondition that the input fields are continuous on all non-$\mathcal{B}$ values, yet have discrete values, this means every input field consists of a union of open sets mapping to constant values, plus a complementary space mapping to $\mathcal{B}$. Since a finite intersection of open sets is open, the intersections of these open sets is also a collection of open sets, with a complementary space of all points where at least one input field maps to $\mathcal{B}$. Since the output of any `df` operator at any event m is affected only by the values of its inputs at m, this means that each open set of constant-valued inputs maps to an open set of constant-valued outputs, plus a complementary space mapping to $\mathcal{B}$, satisfying continuity-preservation.

Consistency is shown the same way as in Lemma 5. All ϵ-approximation sequences must approximate f_o on subspace $f_o^{-1}(\mathbb{V} - \mathcal{B})$ because it is continuous on this space. For the complementary space $\bigcup_i f_i^{-1}(\mathcal{B})$, f_o must also always be approximable because it holds the constant value $\mathcal{B}$ on a finite union of subspaces of approximable fields. □

Lemma 7 *Any built-in operator* `s` *is consistent and continuity-preserving.*

PROOF. As with `df` in Lemma 6, if `s` is continuous on the non-boundary portions of its first input $f_1^{-1}(\mathbb{V} - \mathcal{B})$, then it must consist of a union of open sets mapping to constant values, plus a complementary space mapping to $\mathcal{B}$. Recall also that by the semantics of field calculus, all inputs of `s` are all required to have the same domain. Since the output of any `s` operator at any event m is affected only by the values of its inputs at m, this means that the output for each such open set must obtain its values from an open set of some other input field f_i, where $i > 1$. Since f_i must be continuous on $f_i^{-1}(\mathbb{V} - \mathcal{B})$, so must its intersection with an open set. Every other point outside of this set must be mapped to $\mathcal{B}$, either due to having $\mathcal{B}$ in the first input or else having $\mathcal{B}$ in the selected input, thus satisfying continuity-preservation.

Consistency is shown the same way as in Lemma 5. All ϵ-approximation sequences must approximate f_o on subspace $f_o^{-1}(\mathbb{V} - \mathcal{B})$ because it is continuous on this space. For the complementary space $f_o^{-1}(\mathcal{B})$, f_o must also always be approximable because it holds the constant value $\mathcal{B}$ on a finite intersection and union of subspaces of approximable fields. □

Lemma 8 *Any built-in operator* `d` *is consistent and continuity-preserving.*

PROOF. By definition, a `d` operator maps a collection of non-intersecting open intervals from the reals to constant values. Let us designate this collection C, and members of the collection $c \in C$.

Consider any point $m \in M$. If $f_1(m) \in c$ for some $c \in C$, then the continuity precondition on f_1 implies that there is also some ϵ such that $f_1(m') \in c$ for every point m' in an ϵ-ball around m. This, in turn, implies that f_o is also continuous at that point, because every point in the ϵ-ball will map to the same constant value. Complementarily, any point m such that $f_1(m) \notin C$ maps to $\mathcal{B}$ in f_o and is excluded from requirement for continuity. Thus, continuity-preservation is satisfied.

For every $c \in C$, all ϵ-approximation sequences must approximate f_o on subspace $f_1^{-1}(c)$ because the continuity precondition on f_1 implies that $f_1^{-1}(c)$ is an open set, and f_o has a constant value on that space. Consistency thus holds for f_o for the union of such subspace, subspace $f_1^{-1}(C)$. Likewise, by the precondition of approximability on f_1, consistency holds for subspace $f_1^{-1}(\mathcal{B})$. This leaves only subspace $field_1^{-1}(\mathbb{R} - C)$, which all map to $\mathcal{B}$, and since it is constant-valued and the complement of an approximable subspace, consistency is satisfied overall. □

Lemma 9 *Branch operator* `if` *is consistent and continuity-preserving.*

PROOF. The `if` operator is equivalent to an `s` built-in except that its branch expressions are evaluated with respect to the subspaces $f_1^{-1}(\textit{true})$ and $f_1^{-1}(\textit{false})$ respectively (i.e., the domain of the evaluation environment is reduced). Since these are open subspaces of the environment's domain M, the approximability and continuity properties of e are not affected by the domain reduction. Thus, if the branch expressions would have conformed with the preconditions for M they will also conform with the preconditions for the branch subspaces. The output field is then assembled piecewise identically to an `s` operation, and is consistent and continuity-preserving on $f_o^{-1}(\mathbb{V} - \mathcal{B})$ by the same reasoning. □

Lemma 10 *Function call operator f is eventually consistent and eventually continuity-preserving.*

PROOF. Consider a function definition `f`; either `f` contains other function calls or it does not. If it does not, then evaluating `f` is equivalent to evaluating a composition of operators satisfying Lemma 3 (adjoining the function arguments to the environment and substituting `sense` functions for variable references). Thus the desired consistency and continuity properties hold.

If `f` does contain function calls, then the properties hold if they hold for all of the function calls within `f` (meaning that once again Lemma 3 can apply). Since GPI-calculus does not allow recursion, it must be the case that these dependencies between function definitions can be arranged in a finite directed acyclic graph. The nodes of such a graph may then be ordered, such that each subsequent node only depends on nodes before it in the order. Since the set is finite, there must be at least one node that has no dependencies and thus forms a base case for induction showing that the properties hold for all function calls. □

Lemma 11 *Operator `GPI` is eventually consistent and eventually continuity-preserving.*

PROOF. Following the semantics of field calculus to interpret the `GPI` algorithm given in Section 3 of the main text, the first element of the tuple computed in the `rep` statement implements a computation of distance via the triangle inequality (`nbr-range` is a metric, and a metric multiplied by a continuous positive scalar function is still a metric).

Thus, if there is a spatial section S_M such that the values of all of the inputs do not change at any device on $T^+(S_M)$, then since we assume that manifolds have finite diameter, it must be the case that all the distance estimates (first values of the `distance-integral` tuple) eventually converge to a continuous field of distance estimates. The values of the integral are co-computed with the values of the distance estimates, so they too will stop changing once the inputs have stopped changing. Thus it must be possible to choose a spatial section S'_M on which values of `distance-integral` do not change at any device.

Note also that due to the use of `min-hood`' in computing the triangle inequality, any point in the field of distances with more than one shortest path leading to it will be replaced by a $\mathcal{B}$. This eliminates only a set of measure zero: because the distance function is continuous, its gradient cannot be discontinuous on a space of more than measure zero. The set eliminated is, however, precisely the set of points for which the gradient of the distance field is not continuous.

This is important because this is also the set of points on which the integral calculated in the second element of the `distance-integral` tuple might not be continuous. Consider, for example, the case shown in Figure 11, in which a hole causes there to be two very different shortest paths to a point. The integrals computed along such paths may be very different indeed, necessarily creating a discontinuity in the value of the integral and hence the output of `GPI`. This type of problem can also come from other sources besides topological complexity: similar patterns (and failures) can be caused by `if` statements, distortions in the distance measure, or the shape of the source—anything that can cause a discontinuity in the gradient.

Since `min-hood`' ensures that such points are $\mathcal{B}$, however, they are eliminated from the region on which we must establish continuity. If we instead consider any m with precisely one shortest path to the `source` region, then because both integrand and the gradient of the computed distance field are continuous on this path, it must be the case that for any given ϵ, there must be a δ such that the open set of events within

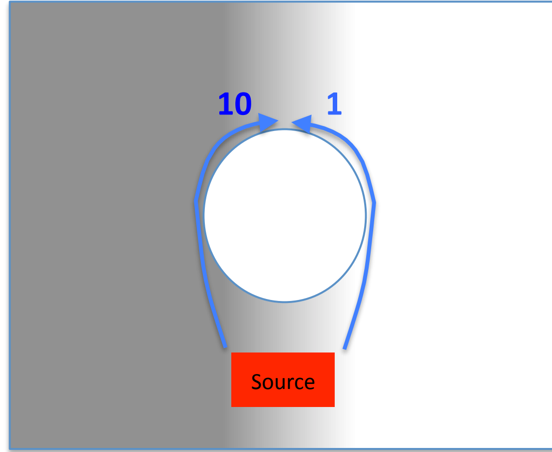


Fig. 11. The field of path integral values created by `GPI` can be discontinuous at points reached by two different shortest paths, as in this example where the left path around the hole goes through a region where the integrand is much higher than on the right path, resulting in discontinuity between a value of 10 from the left and 1 from the right.

distance δ have integral values that are less than ϵ different, and thus the field of integrals output by `GPI` is continuous on $f_o^{-1}(\mathbb{V} - \mathcal{B}) \cap T^+(S_M)$. Because it is continuous on this space, it must also be approximable; the complementary space of $\mathcal{B}$ values must also be approximable, as it is a union of the $\mathcal{B}$ values of the inputs (which must be approximable) plus the measure zero set of $\mathcal{B}$ events added from computation of the distance function. □

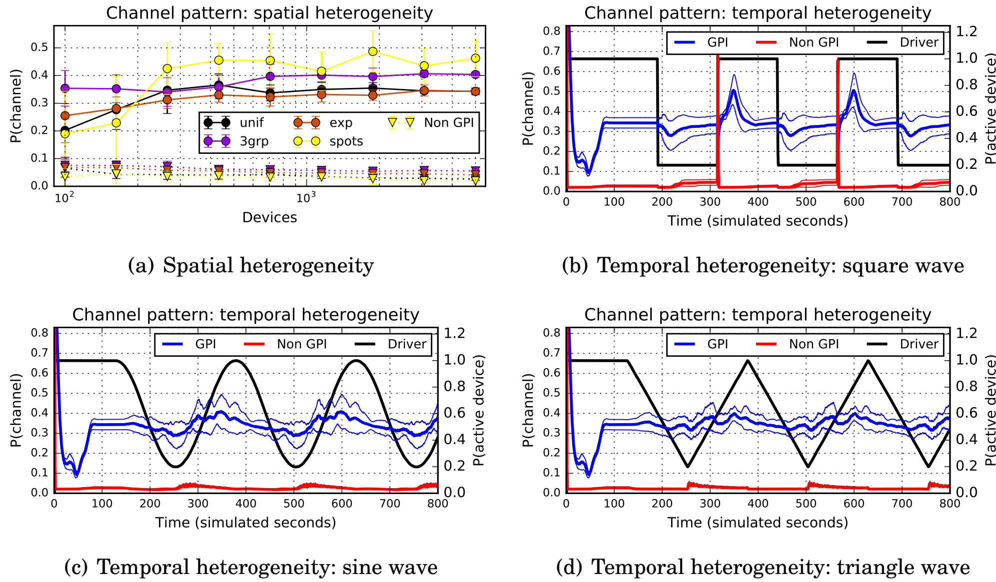
Theorem 1 (Eventual Consistency of GPI-calculus) *GPI-calculus programs are eventually consistent for all environments e that are continuous on $e^{-1}(\mathbb{V} - \mathcal{B})$.*

PROOF. By Lemmas 2 and 4–11, we know that every operator in GPI-calculus is eventually consistent and eventually continuity-preserving. Because recursion is prohibited, it must be the case that the number of operators evaluated in the evaluation of a GPI-calculus program must be finite for any given ϵ-approximation. Furthermore, since `if` is the only method of branching evaluations, it must thus be the case that in any approximation sequence there is some i, after which no $\epsilon_{j>i}$-approximation evaluates an operator instance that is not also evaluated in some prior ϵ-approximation, considering any instance in which an operator instance is not evaluated to be an evaluation with null domain.

Given the precondition of an environment continuous on $e^{-1}(\mathbb{V} - \mathcal{B})$, we thus satisfy the conditions of Lemma 3 and have that any GPI-calculus program must be eventually consistent. □

B. ADDITIONAL DETAILS ON RESILIENCE TO HETEROGENEITY RESULTS

This supplementary section presents further details on the application results presented in Section 5. In particular, we show a detailed comparison between the behavior of the GPI-calculus and non-GPI-calculus versions of the algorithms for the space and time heterogeneity experiments, along with variation information elided from temporal heterogeneity graphs.



(a) Spatial heterogeneity

(b) Temporal heterogeneity: square wave

(c) Temporal heterogeneity: sine wave

(d) Temporal heterogeneity: triangle wave

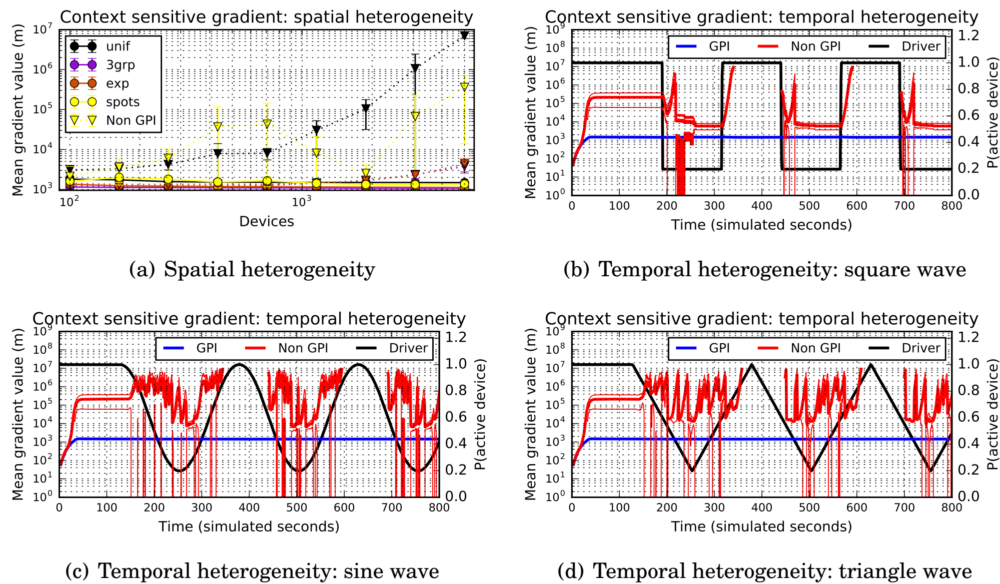
Fig. 12. Additional details of experimental data on wireless sensor network scenario confirming our predictions for GPI-calculus. (a) compares GPI performance of Figure 8(c) with the non-GPI version (triangles). (b,c,d) compare GPI performance of Figure 8(d) (blue) with the non-GPI version (red) for each of the three density driver waves (black). All graphs show mean and ± 1 standard deviation for both GPI and non-GPI versions. As in the other tests in Figure 8, the non-GPI is disrupted by the fragility of floating point equality operations in every experiment. Results are the average of 10 simulation runs.

B.1. Distance-Based Patterns in a Wireless Sensor Network

Figure 12 shows additional result details for the distance-based pattern in a wireless sensor network, conforming with those presented in Section 5. The GPI-calculus algorithm stabilizes regardless the exact distribution of devices in spaces. The differences between the configurations that can be seen in Figure 12(a) are due to the chosen metric: since the channel has a precise location in space, configurations where more nodes are located in such area (see e.g., Figure 7(b)) may return a higher probability value, though in both cases the pattern has stabilized correctly. The GPI-calculus version of the algorithm is also resilient to dynamic changes in device density. For any tested driver, after a transient, the algorithm recovers to the expected values. The square wave causes the widest shift from the expected value immediately after many new devices join the system, but this effect is quickly mitigated. In all cases, however, the non-GPI version is disrupted by the fragility of floating point equality operations and fails to reliably build a connection between source and destination, as reflected in the extremely low $P(channel)$ values.

B.2. Context-Sensitive Distance in Urban Traffic Steering

Figure 13 shows additional result details for the context-sensitive distance in an urban traffic steering scenario, conforming with those presented in Section 5. The GPI-calculus version of the algorithm is extremely stable when compared to the naive non-GPI alternative, which exhibits behavior that is extremely erratic and sensitive to changes in density and distribution of devices both in space and time if compared to the GPI-calculus version. The variation is so high, in fact, that we have chosen to use logarithmic plots in Figure 13 in order to allow changes to be visible across the whole



(a) Spatial heterogeneity

(b) Temporal heterogeneity: square wave

(c) Temporal heterogeneity: sine wave

(d) Temporal heterogeneity: triangle wave

Fig. 13. Additional details of experimental data on urban traffic steering scenario confirming our predictions for GPI-calculus. (a) compares GPI performance of Figure 10(c) with the non-GPI version (triangles). (b,c,d) compare GPI performance of Figure 10(d) (blue) with the non-GPI version (red) for each of the three density driver waves (black). All graphs show mean and ± 1 standard deviation for GPI and non-GPI versions. As in the other tests in Figure 10, the non-GPI version is strongly affected by the number of devices populating the scenario, and also fails to effectively stabilize. Results are the average of 10 simulation runs.

scale of behaviors. As can be readily seen, the fluctuations in time of the values of the GPI-calculus version of the algorithm, that can be observed in Figure 10, are dwarfed by the wider changes of the non-GPI naive alternative.

REFERENCES

H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. 1999. *Amorphous Computing*. Technical Report AIM-1665. MIT.

Alexander Artikis, Marek J. Sergot, and Jeremy V. Pitt. 2009. Specifying norm-governed computational societies. *ACM Trans. Comput. Log.* 10, 1 (2009). DOI:http://dx.doi.org/10.1145/1459010.1459011

Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. 2007. Meld: A Declarative Approach to Programming Ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*. IEEE Press, Piscataway, NJ, USA, 2794–2800.

Jacob Beal. 2005. Programming an Amorphous Computational Medium. In *Unconventional Programming Paradigms*, Jean-Pierre Banatre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel (Eds.). Lecture Notes in Computer Science, Vol. 3566. Springer Berlin Heidelberg, Berlin, 121–136. DOI:http://dx.doi.org/10.1007/11527800_10

Jacob Beal. 2010. A Basis Set of Operators for Space-Time Computations. In *Proceedings of the 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW '10)*. IEEE Computer Society, Washington, DC, USA, 91–97. DOI:http://dx.doi.org/10.1109/SASOW.2010.21

Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2013. Organizing the Aggregate: Languages for Spatial Computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, Marjan Mernik (Ed.). IGI Global, Hershey, PA, Chapter 16, 436–501. DOI:http://dx.doi.org/10.4018/978-1-4666-2092-6.ch016

Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate Programming for the Internet of Things. *IEEE Computer* 48, 9 (2015), 22–30.

Jacob Beal, Kyle Usbeck, and Brett Benyo. 2013. On the Evaluation of Space-Time Functions. *Comput. J.* 56, 12 (2013), 1500–1517. DOI:http://dx.doi.org/10.1093/comjnl/bxs099

Jacob Beal and Mirko Viroli. 2014. Building Blocks for Aggregate Programming of Self-Organising Applications. In *Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8-12, 2014*. IEEE, Piscataway, NJ, USA, 8–13. DOI:http://dx.doi.org/10.1109/SASOW.2014.6

Jacob Beal, Mirko Viroli, and Ferruccio Damiani. 2014. Towards a Unified Model of Spatial Computing. In *7th Spatial Computing Workshop (SCW 2014)*. AAMAS 2014, Paris, France.

Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2016. Self-adaptation to Device Distribution Changes. In *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2016)*. IEEE, Piscataway, NJ, USA.

William Butera. 2002. *Programming a Paintable Computer*. Ph.D. Dissertation. MIT, Cambridge, MA, USA.

Lauren Clement and Radhika Nagpal. 2003. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*.

Daniel Coore. 1999. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. Ph.D. Dissertation. MIT, Cambridge, MA, USA.

Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. 2005. Mobile data collection in sensor networks: The TinyLime middleware. *Elsevier Pervasive and Mobile Computing Journal* 4 (2005), 446–469.

Ferruccio Damiani and Mirko Viroli. 2015. Type-based Self-stabilisation for Computational Fields. *Logical Methods in Computer Science* 11, 4 (2015), 1–53. DOI:http://dx.doi.org/10.2168/LMCS-11(4:21)2015

Ferruccio Damiani, Mirko Viroli, and Jacob Beal. 2016. A type-sound calculus of computational fields. *Science of Computer Programming* 117 (2016), 17–44.

Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. 2015. Code Mobility Meets Self-organisation: a Higher-order Calculus of Computational Fields. In *Proceedings of FORTE 2015*. Springer International Publishing, Berlin, 113–128.

Shlomi Dolev. 2000. *Self-Stabilization*. MIT Press, Cambridge, MA.

Bradley R. Engstrom and Peter R. Cappello. 1989. The SDEF programming system. *J. Parallel and Distrib. Comput.* 7, 2 (1989), 201 – 231.

Fathiyeh Faghih and Borzoo Bonakdarpour. 2015. SMT-Based Synthesis of Distributed Self-Stabilizing Systems. *ACM Trans. Auton. Adapt. Syst.* 10, 3, Article 21 (Oct. 2015), 26 pages. DOI:http://dx.doi.org/10.1145/2767133

JoseLuis Fernandez-Marquez, Giovanna Marzo Serugendo, Sara Montagna, Mirko Viroli, and JosepLluis Arcos. 2013. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing* 12, 1 (2013), 43–67. DOI:http://dx.doi.org/10.1007/s11047-012-9324-y

Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. 2002. *Computational models for integrative and developmental biology*. Technical Report 72-2002. Univerite d'Evry, LaMI.

Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. 2005. Computations in Space and Space in Computations. In *Unconventional Programming Paradigms*, Jean-Pierre Bantre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel (Eds.). Lecture Notes in Computer Science, Vol. 3566. Springer Berlin Heidelberg, Berlin, 137–152. DOI:http://dx.doi.org/10.1007/11527800_11

T.A. Henzinger. 1996. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*. IEEE, Piscataway, NJ, USA, 278–292. DOI:http://dx.doi.org/10.1109/LICS.1996.561342

Anup K. Kalia and Munindar P. Singh. 2015. Muon: designing multiagent communication protocols from interaction scenarios. *Autonomous Agents and Multi-Agent Systems* 29, 4 (2015), 621–657. DOI:http://dx.doi.org/10.1007/s10458-014-9264-2

H Kestelman. 1960. *Modern Theories of Integration* (2nd. rev. ed. ed.). Dover, New York, Chapter "Lebesgue Integral of a Non-Negative Function" and "Lebesgue Integrals of Functions Which Are Sometimes Negative." (Chapter 5-6), 113–160.

Attila Kondacs. 2003. Biologically-inspired Self-Assembly of 2D Shapes, Using Global-to-local Compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

C. Lasser, J.P. Massar, J. Miney, and L. Dayton. 1988. Starlisp Reference Manual. (1988).

Victor Lesser, Keith Decker, Thomas Wagner, Norman Carver, Alan Garvey, Bryan Horling, Daniel Neiman, Rodion Podorozhny, M Nagendra Prasad, Anita Raja, and others. 2004. Evolution of the GPGP/-TAEMS domain-independent coordination framework. *Autonomous agents and multi-agent systems* 9, 1-2 (2004), 87–143.

Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Bruce MacLennan. 1990. *Continuous Spatial Automata*. Technical Report Department of Computer Science Technical Report CS-90-121. University of Tennessee, Knoxville.

Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2002. TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 131–146. DOI:http://dx.doi.org/10.1145/844128.844142

Ashok U. Mallya and Munindar P. Singh. 2007. An algebra for commitment protocols. *Autonomous Agents and Multi-Agent Systems* 14, 2 (2007), 143–163. DOI:http://dx.doi.org/10.1007/s10458-006-7232-1

Marco Mamei and Franco Zambonelli. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. on Software Engineering Methodologies* 18, 4 (2009), 1–56. DOI:http://dx.doi.org/10.1145/1538942.1538945

Radhika Nagpal. 2001. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. Ph.D. Dissertation. MIT, Cambridge, MA, USA.

Ryan Newton and Matt Welsh. 2004. Region Streams: Functional Macroprogramming for Sensor Networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*. ACM, New York, NY, USA, 78–87.

Andrea Omicini, Alessandro Ricci, and Mirko Viroli. 2008. Artifacts in the A&A Meta-Model for Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems* 17, 3 (June 2008). DOI:http://dx.doi.org/10.1007/s10458-008-9053-x

H. Van Dyke Parunak, Sven Brueckner, Robert S. Matthews, and John A. Sauter. 2005. Pheromone learning for self-organizing agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part A* 35, 3 (2005), 316–326. DOI:http://dx.doi.org/10.1109/TSMCA.2005.846408

Danilo Pianini, Jacob Beal, and Mirko Viroli. 2016. Improving Gossip Dynamics Through Overlapping Replicates. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science)*, Alberto Lluch Lafuente and José Proença (Eds.), Vol. 9686. Springer, 192–207. DOI:http://dx.doi.org/10.1007/978-3-319-39519-7_12

Danilo Pianini, Sara Montagna, and Mirko Viroli. 2013. Chemical-oriented Simulation of Computational Systems with Alchemist. *Journal of Simulation* 7, 3 (2013), 202–215. DOI:http://dx.doi.org/10.1057/jos.2012.27

Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: Practical Aggregate Programming. In *ACM Symposium on Applied Computing 2015*. ACM, New York, NY, USA, 1846–1853.

F Raimbault and D Lavenier. 1993. ReLaCS for Systolic Programming. In *Int'l Conf. on Application-Specific Array Processors*. 132–135.

Edwin F. Taylor and John Archibald Wheeler. 1992. *Spacetime Physics: Introduction to Special Relativity* (2nd ed. ed.). W. H. Freeman & Company, Gordonsville, VA.

Matthew E Taylor, Manish Jain, Christopher Kiekintveld, Jun-young Kwak, Rong Yang, Zhengyu Yin, and Milind Tambe. 2011. Two decades of multiagent teamwork research: past, present, and future. In *Collaborative Agents-Research and Development*. Springer, 137–151.

Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2015a. Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields. In *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2015)*. IEEE, Piscataway, NJ, USA, 81–90.

Mirko Viroli and Ferruccio Damiani. 2014. A Calculus of Self-stabilising Computational Fields. In *Coordination Languages and Models*, eva Kühn and Rosario Pugliese (Eds.). LNCS, Vol. 8459. Springer-Verlag, 163–178. DOI:http://dx.doi.org/10.1007/978-3-662-43376-8_11 Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June.

Mirko Viroli, Danilo Pianini, Sara Montagna, Graeme Stevenson, and Franco Zambonelli. 2015b. A coordination model of pervasive service ecosystems. *Science of Computer Programming* 110 (2015), 3 – 22. DOI:http://dx.doi.org/10.1016/j.scico.2015.06.003

Mirko Viroli, Danilo Pianini, Alessandro Ricci, Pietro Brunetti, and Angelo Croatti. 2015c. Multi-agent Systems Meet Aggregate Programming: Towards a Notion of Aggregate Plan. In *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, Qingliang Chen, Paolo Torroni, Serena Villata, Jane Hsu, and Andrea Omicini (Eds.). Lecture Notes in Computer Science, Vol. 9387. Springer International Publishing, 49–64. DOI:http://dx.doi.org/10.1007/978-3-319-25524-8_4

Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. 2004. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press.

Daniel Yamins. 2007. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. Ph.D. Dissertation. Harvard, Cambridge, MA, USA.

Yong Yao and Johannes Gehrke. 2002. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record* 31 (2002), 2002.

Li-Hsing Yen, Jean-Yao Huang, and Volker Turau. 2016. Designing Self-Stabilizing Systems Using Game Theory. *ACM Trans. Auton. Adapt. Syst.* 11, 3, Article 18 (Sept. 2016), 27 pages. DOI: http://dx.doi.org/10.1145/2957760

Received December, 2016; revised XXXX; accepted XXXX