From distributed coordination to field calculus and aggregate computing

(Article begins on next page)

26 April 2024

From Distributed Coordination to Field Calculus and Aggregate Computing[1]

Mirko Viroli[a], Jacob Beal[b], Ferruccio Damiani[c], Giorgio Audrito[c], Roberto Casadei[a], Danilo Pianini[a]

[a]*Alma Mater Studiorum—Università di Bologna, Italy*
{mirko.viroli,roby.casadei,danilo.pianini}@unibo.it
[b]*Raytheon BBN Technologies, USA*
jakebeal@ieee.org
[c]*Università di Torino, Italy*
ferruccio.damiani@unito.it,giorgio.audrito@unito.it

Abstract

Aggregate computing is an emerging approach to the engineering of complex coordination for distributed systems, based on viewing system interactions in terms of information propagating through collectives of devices, rather than in terms of individual devices and their interaction with their peers and environment. The foundation of this approach is the distillation of a number of prior approaches, both formal and pragmatic, proposed under the umbrella of field-based coordination, and culminating into the *field calculus*, a universal functional programming model for the specification and composition of collective behaviours with equivalent local and aggregate semantics. This foundation has been elaborated into a layered approach to engineering coordination of complex distributed systems, building up to pragmatic applications through intermediate layers encompassing reusable libraries of program components. Furthermore, some of these components are formally shown to satisfy formal properties like self-stabilisation, which transfer to whole application services by functional composition. In this survey, we trace the development and antecedents of field calculus, review the field calculus itself and the current state of aggregate computing theory and practice, and discuss a roadmap of current research directions with implications for the development of a broad range of distributed systems.

Keywords: distributed systems, aggregate computing, field calculus, spatial computing

September 27, 2019

1. Introduction

As computing devices continue to become cheaper and more pervasive, the complexity of the distributed systems that run our world continues to increase. Over the past several decades, we have moved from many people sharing a single computer to a computer for each person to many, mostly embedded and minimal-interface computing devices for each person. The only way to effectively engineer and coordinate the operation of such systems is to program and operate in terms of *aggregates* of devices rather than attempting to micromanage each individual device. Moreover, as devices become more numerous, smaller, and more embedded, decentralisation brings new opportunities as well as new challenges—not only in terms of pervasive sensing/actuation/computation abilities, but also of increasing requirements for resilience, efficiency, privacy, sustainability, and other non-functional requirements.

Aggregate computing is an emerging approach, developed significantly within the coordination models and languages research community, that embraces this environment, and with the core idea of functionally composing collective behaviours to achieve effective and resilient complex behaviours in dynamic networks. As an example, to provide smart-mobility services for pedestrians in smart-cities [1], with aggregate computing one might first program a distributed building block to estimate crowded areas (e.g., reusing core library blocks to compute distances, elect distributed leaders, and collect information from regions to target nodes), and then functionally stack on top a service to alert people in crowded areas, another to disperse them by suggesting steering directions, and yet another to guide people to points of interest while circumventing those crowded areas.

Aggregate computing builds from a foundation of the *field calculus*, a functional programming model for the specification and composition of collective behaviours with formally equivalent local and aggregate semantics. Atop this foundation, a layered approach has been constructed to engineering coordination of complex distributed systems in contexts such as smart-cities and smart-environments, robot/drone swarms, and tactical networks. This has been achieved by first considering challenges of resilience, then pragmatism in the form of reusable libraries capturing common coordination patterns, and finally applications across a number of different domains. As the research on aggregate computing is becoming rather multi-faceted, we also envision a variety of research directions of high importance for distributed systems and specifically for coordination models and languages, in theory, in engineering methods and tools, and in applications.

In this survey, an extended version of [2], we present a discussion of the past, present, and future of aggregate computing (Figure 1). The paper expands on [2] through a broader coverage of related works (including the pictorial overview in Figure 2), formal properties studied, and current/future works in several areas (platforms, security, applications); together with the addition of a section (Section 3.3) presenting the denotational semantics for the first-order field calculus, obtained by simplifying the semantics previously published for the higher-order
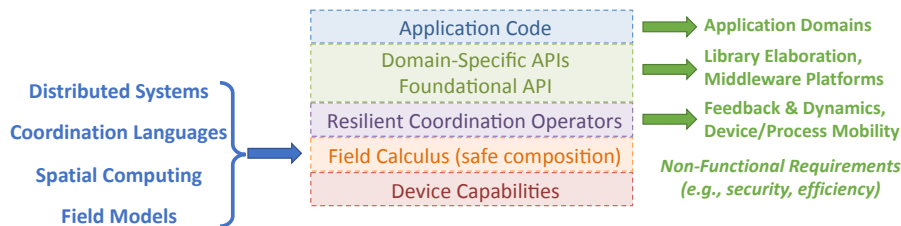
Figure 1: This survey reviews the development of field calculus from its antecedents (left), the current state of aggregate computing theory and practice as layered abstractions based on field calculus (middle), and current research directions on stemming from field calculus and aggregate programming (right).

version of the calculus.

Section 2 begins by tracing the development of aggregate computing through its antecedents both in coordination research and in other areas, culminating in the development of the field calculus. Section 3 then presents a detailed review of field calculus, the formal foundation of aggregate computing, as well as examples of aggregate programming. Section 4 discusses the current state of the rest of aggregate computing theory and practice across its various abstraction layers. Section 5 presents a roadmap of current research directions on top of field calculus and with respect to challenges in coordination models and languages. Finally, Section 6 summarises and concludes the paper.

2. Coordination, Self-Organisation, and Fields

In this section, we review and discuss the conceptual, but also technical and technological, path that has brought traditional coordination models for parallel computing, step-by-step to address the complexity of self-organising, large-scale deployed systems (Section 2.1 and Section 2.2). We then describe the emergence of field-based coordination (Section 2.3), and how, through the interaction with research falling under the umbrella of space-based computation models (Section 2.4), this path has led to the development of the field calculus and aggregate computing. A pictorial overview of the antecedents of field calculus and aggregate computing discussed in this section is provided in Figure 2.

2.1. Coordination towards Self-organisation

One of the key threads of antecedent research begins with simple coordination of parallel activities, then moves towards increasing intelligence in coordination and distribution into increasingly complex self-organising distributed coordination systems.

3

Linda [3]

Generative communication

JavaSpaces [4]
TSpaces [5]

Tuple-based coordination models

Shared Prolog [6]
Law-Gov. Inter. [7]
MARS [8]
ReSpecT [9]

Programmable coordination rules

LogOp [10], Scope [11]
Klaim [12], Lime [13]

Distributed, mobile coordination

$\sigma\tau$-Linda [14]
SwarmLinda [15]
Biochemical TSs [16]

Biology

Chemistry

Ecology

Self-organising coordination

Physics

Field-based coordination

TOTA [17]

Hood [23]
Abstract Regions [24]
Paintable comp. [25]
Meld [26]

Network abstraction

TinyDB [34]
Cougar [35]
TinyLime [36]
Regiment [37]

Spatial streams

GPL [21]
OSL [22]

Spatial pattern languages

Spatial Computing

Proto [18]
MGS [19, 20]

E-institutions [32]
$\mathcal{M}oise^+$ [33]

MAS coordination

SCEL [27]
Carma [28]
Helena [29]
DEECo [30]
GCM/ProActive [31]

Collective adaptive systems
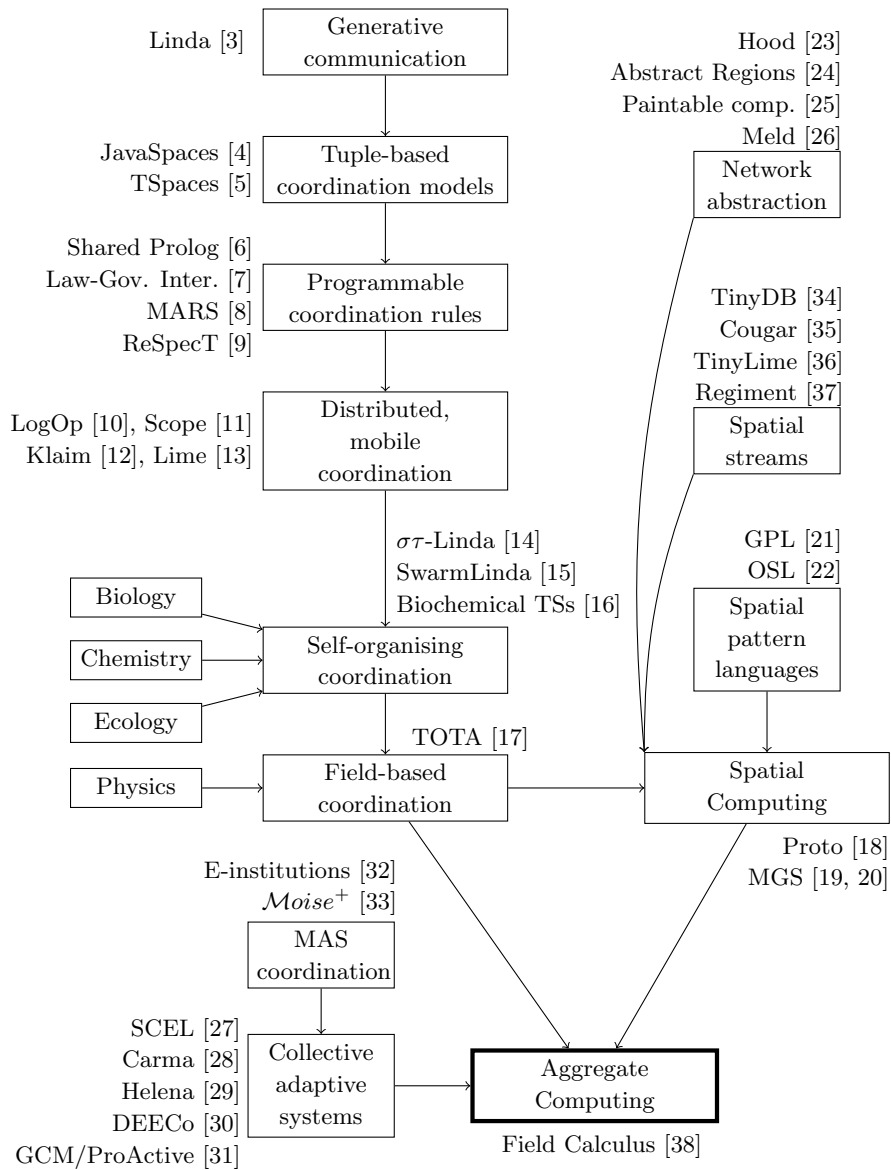
Aggregate Computing

Field Calculus [38]

Figure 2: Overview of research threads leading from coordination to field calculus and aggregate computing, highlighting some bibliographic references. This summary—by no means exhaustive—provides key highlights for the perspective discussed in this paper.

Generative communication

Coordination models are rooted in the idea that interaction among multiple, independent, and autonomous software systems (e.g., processes, components, and so on, somewhat generically called *agents* henceforth) could be conceived and designed as a space orthogonal to pure computation. Historically, many coordination models reify this idea into a concept of shared data space, working as a whiteboard, where processes of a parallel computing system can write and read information [39], enabling so-called *generative communication*. Linda [3] is broadly recognised as the ancestor of a number of approaches to generative communication falling under the umbrella of tuple-based coordination models. The foundational idea of Linda was to have processes (on a centralised system) share information and synchronise by writing and retrieving, with a suspensive semantics (the requester is blocked until the query is satisfiable), data in the form of an ordered collection of possibly-heterogeneous knowledge chunks, i.e., tuples, from a shared (tuple-)space. Such data could be retrieved associatively, by querying through partial representations of the structure and content matching the desired piece of data (tuple template). The consequence is twofold: *(i)* decoupling in communication is strongly promoted, since no information about the sender, the space itself, and the tuple insertion time is required in order for communication to happen; and *(ii)* coordination is still possible in environments where information is vague, incomplete, inaccurate, or not entirely specified, due to the possibility of synchronising over a partial representation of knowledge.

Programmable coordination rules

The vision of tuple-based coordination as a shared knowledge repository used for agent coordination is further promoted by logical tuple-space models, where software agents coordinate through first-order logic tuples, and tuple spaces can be programmed as first-order logic theories. A prominent example of such approach is Shared Prolog [6], a framework for writing multi-processor Prolog systems. More generally, this view promotes the idea of equipping the shared space with some form of "intelligence", e.g., in the form of an application logic that can manipulate data in the shared space and the way that it can be accessed. Several Linda-inspired approaches tackle this issue by enabling programmability at the tuple-space level in order to express rules of coordination, and hence, pushing forward a notion of expressiveness of the coordination media [40]. Among them, we find Law-Governed Interaction [7], which structures the coordination logic within groups of agents by explicit "rules of engagement"; MARS [8], whose tuple spaces can be programmed with stateful "reaction objects" triggered upon access patterns; and ReSpecT [9], where logic specification tuples map events to transactional sequences of reactions, which are primitive invocations of logic-based computations.

Distribution

All these approaches, however, do not explicitly focus on distributed systems, but on the coordination of centralised local components. As software components become spread across the system network, so multiple tuple spaces

5

can be distributed across the system environment, enabling distributed coordination abstractions, featuring mechanisms for event-based interactions, timing, and advanced data representation. This is the case with industrial systems like JavaSpaces [4], an API for distributed coordination through persistent, shared spaces of objects, and TSpaces [5], which combines Linda-like spaces with asynchronous messaging. Some middlewares take the approach a step further, by dealing with location and mobility, and enabling expression of dynamic environment topologies in a distributed setting, thus paving the way towards application of coordination models to pervasive computing system scenarios. For instance, LogOp [10] extends basic Linda with coordination primitives for dynamically accessing multiple distributed tuple spaces based on logical expressions. Scope [11] leverages distributed broadcasts for tuple placement and migration. In Lime [13], mobile agents communicate with each other through "transiently shared tuple spaces" whose content is dynamically reconfigured based on the set of co-located agents. Another example is Klaim [12], which exposes a programming paradigm of mobile processes and data where explicit *localities* regulate the interaction protocol of located processes and types characterise the intention of processes with respect to specific localities.

Self-organising coordination

As coordination abstractions of various sorts (e.g., tuple spaces, channels, coordination artefacts [41, 42]) are available in distributed settings, one is directly faced with the problem of dealing with openness (hence, unexpectedness of environment changes, faults, and interactions), large scale (possibly a huge number of agents and coordination abstractions to be managed), and intrinsic adaptiveness (such as the ability to intercept relevant events and react to them to guarantee overall system resilience). This calls for an approach of *self-organising coordination* [43], where coordination abstractions handle "local" interactions only (and typically use stochastic mechanisms to keep the coordination process always "up and running"), such that global and robust patterns of correct coordination behaviour can emerge—achieved by trading off by-design adaptiveness with inherent, automatic forms of adaptiveness.

Coordination models following this approach typically take their inspiration from complex natural systems (from physics through chemistry, all the way to ethology) and attempt to reuse the foundational mechanisms of such systems. A primary source of inspiration for these systems is to be found in biology (social animals, and insects in particular), whose foraging techniques inspire mechanisms to regulate coordination [44, 15, 45]. For instance, SwarmLinda [15] is a tuple-based middleware that brings the collective intelligence displayed by swarms of ants to computational mechanisms aimed at guaranteeing efficient retrieval of tuples. Tuples are handled as forms of pheromones or items that ants (agents) continuously and opportunistically relocate. Chemical inspiration is used in [46, 16] to regulate the "activity level" of tuples, which drives the likelihood of their retrieval as well as their propagation rate. Ecological inspiration is instead used in [47] to inject competition, composition, and disposal behaviour in the context of coordination of pervasive computing services.

6

2.2. Multi-Agent and Collective Adaptive Systems

The research line of *Multi-Agent Systems (MAS)* [48] inherently acknowledges the key role of coordination [49] by focussing on the macro level of systems of interacting autonomous agents. One key coordination challenge is to make agents cooperate despite conflicting goals, e.g., through consistent multi-agent planning and proper negotiation. The survey in [50] provides an account of recent progress in distributed multi-agent coordination in the areas of consensus, formation control, optimisation, task assignment, and estimation.

Additionally, MAS research recognises the importance that the organisational dimension [51] assumes in the realisation of system-level behaviour. Indeed, the function of structure and order is to regulate interactions so as to achieve static or dynamic goals. This significance has motivated the emergence of frameworks and linguistic approaches (grouped under the notion of *organisation-oriented programming* [52]) to model the organisational dimension of MAS, such as e-institutions [32] and $\mathcal{M}oise^+$ [33]. The perspective of self-organisation is particularly relevant in MAS [53], as it provides a way to deal with change in the environment and system itself.

A closely related branch of research that focuses on macro-level behaviour, especially in large-scale dynamic scenarios, is that of *Collective Adaptive Systems (CAS)* [54]. Decentralisation of control, non-synchronised operation, and opportunistic interaction are often essential in this context to deal with the scale and changes in both the system structure and environment. In this research area, it is common to consider large, dynamic groups of devices as first-class abstractions – sometimes called *ensembles*, *collectives*, or *aggregates* – and support interaction between (sub-)groups of devices by abstracting from certain details (e.g., networking, or individual logical connections). For instance, in Helena [29], components can dynamically participate in multiple ensembles according to different roles. Similarly, DEECo [30] is another CAS model where components can only communicate by dynamically binding together through ensembles. The GCM/ProActive framework [31] supports the development of large-scale ensembles of adaptable autonomous devices through a hierarchical component model where components have a non-functional membrane and "collective interfaces", and a programming model based on active objects. SCEL [27] is a kernel language to specify the behaviour of autonomic components, the logic of ensemble formation, as well interaction through attribute-based communication (which enables implicit selection of a group of recipients). Carma [28] uses attribute-based communication as well, to coordinate large ensembles of devices via local broadcast operations. In these approaches, the ensemble abstraction is dynamic—in order to cope with change—and hence provides a way to adapt the coordination logic.

2.3. Field-based coordination

Another important natural source of inspiration comes from physics: a number of physics-inspired self-organising coordination systems rely on the notion of "field" (gravitational field, electromagnetic field), which essentially provides

7

a framework to handle (create, manipulate, combine) global-level, distributed data structures.

A notion of *coordination field* (or co-field) was initially proposed in [55] as a means to support self-organisation patterns of agent movement in complex environments: it was used as an abstraction over the actual environment, spread by both agents and the environment itself, and used by agents (which can locally perceive the value of fields) to properly navigate the environment. Based on this idea, the TOTA (Tuples On The Air) tuple-based middleware [17] was proposed to support field-based coordination for pervasive-computing applications. In TOTA each tuple, when inserted into a node of the network, is equipped with a content (the tuple data), a diffusion rule (the policy by which the tuple has to be cloned and diffused around), and a maintenance rule (the policy whereby the tuple should evolve due to events or time elapsing).

The *evolving tuples* model, presented in [56], is an extension to traditional Linda tuple spaces with the goal of supporting resource discovery in a pervasive system, relying on ideas similar to those of TOTA. Evolution is firstly embedded in tuples by adding, to each field of the tuple, a name and a formula that specifies the field behaviour over time. Formulas support the if-then-else construct and arithmetic and boolean operators. Secondly, a new operation `evolve()` is introduced in the tuple space, which is responsible for applying formulas to tuples using contextual information.

One of the first works connecting field-based coordination with formalisation tools typical of coordination models and languages (e.g., process algebras and transition systems) is the $\sigma\tau$-Linda model [14], where agents can inject into the space "processes" that spread, collect and decay tuples, ultimately sustaining fields of tuples.

2.4. Spatial computing approaches: towards the field calculus

More or less independently to the problem of finding suitable coordination models for distributed and situated systems, a number of works addressed similar problems in the more general attempt of building distributed intelligent systems by promoting higher abstractions of spatial collective adaptive systems. Works such as [57, 58, 59, 60] survey from various different viewpoints the many approaches that fall under this umbrella (including also some of the above mentioned coordination models), and which mainly organise in the following categories: methods that simplify programming of a collective by abstracting individual networked devices (e.g., Hood [23], Abstract Regions [24], Butera's "paintable computing" [25], and Meld [26]), spatial patterns and languages (e.g., Growing Point Language [21], geometric patterns in Origami Shape Language [22], self-healing geometries [61], or universal patterns [62]), tools to summarise and stream information over regions of space and time (e.g., TinyDB [34], Cougar [35], TinyLime [36], and Regiment [37]), and finally space-time computing models aiming at the manipulation of data structures diffused in space and evolving with time, e.g., targeting parallel computing (e.g., StarLisp [63], systolic computing [64]) and topological computing (e.g., MGS [19, 20]). Among them, space-time computing models based on

the notion of computational fields were initially proposed in [65] and [18] and implemented in the Proto language [18]. Combining techniques coming from the above approaches and generalising over Proto (which can be considered the archetypal spatial computing language due to its expressiveness and versatility), the field calculus has been proposed as a foundational model for the coordination of computational devices spread in physical environments, also known as *aggregate computing*.

3. Field Calculus

In this section, we review the mathematical core of aggregate computing, the *field calculus* language, together with its most relevant formalisations and properties. We follow the goal of presenting the full spectrum of results achieved while avoiding all deep technical details (which can be accessed if desired from the references provided), though providing a full formalisation of its semantics. First, we present the basic first-order calculus (Section 3.1) together with its operational semantics, typing, basic properties (Section 3.2), and denotational semantics (Section 3.3). We then discuss the behavioural properties that have been studied for field calculus programs (Section 3.4). We conclude by discussing the extension of the calculus to allow for higher-order functions (Section 3.5).

3.1. Basic calculus

The field calculus (FC) was introduced in [66] as a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields:[2] functions over fields, functional composition with fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network.

The field calculus is based on the idea of specifying the aggregate system behaviour of a network of devices, where a dynamic neighbouring relation (which is application-dependent and represents physical or logical proximity) is used to indicate the devices with which one can directly communicate[3]—e.g., in a sensor network, those within the range of a broadcast communication. One such specification is structured as a functional composition of operators that manipulate (evolve, combine, restrict) computational fields.

A key feature of the approach is that a specification can be interpreted either locally or globally. Locally, it can be seen as describing a computation on an individual device, iteratively executed in asynchronous "computation rounds" comprising reception of messages from neighbours, perception of contextual information through sensors, storing local state of computation, computing the local value of fields, and spreading messages to neighbours. Globally, a field

---

[2]This is similar to how λ-calculus [67] captures the essence of functional computation and FJ [68] the essence of class-based object-oriented programming.

[3]A device with no neighbours, e.g., would be one isolated (temporarily or permanently) from the rest of the system.

$$
\begin{array}{lll}
\texttt{P} & ::= & \overline{\texttt{F}}\ \texttt{e} \hfill \text{program} \\[4pt]
\texttt{F} & ::= & \texttt{def}\ \ \texttt{d}(\overline{\texttt{x}})\ \{\texttt{e}\} \hfill \text{function declaration} \\[4pt]
\texttt{e} & ::= & \texttt{x}\ \mid\ \texttt{v}\ \mid\ \texttt{f}(\overline{\texttt{e}})\ \mid\ \texttt{if(e)\{e\}\{e\}}\ \mid \hfill \text{expression} \\
& & \texttt{nbr\{e\}}\ \mid\ \texttt{rep(e)\{(x)=>e\}} \\[4pt]
\texttt{f} & ::= & \texttt{d}\ \mid\ \texttt{b} \hfill \text{function name} \\[4pt]
\texttt{v} & ::= & \ell\ \mid\ \phi \hfill \text{value} \\[4pt]
\ell & ::= & \texttt{c}(\overline{\ell}) \hfill \text{local value} \\[4pt]
\phi & ::= & \overline{\delta}\mapsto\overline{\ell} \hfill \text{neighbouring field value}
\end{array}
$$

Figure 3: Abstract syntax of the field calculus, as adapted from [70]

calculus expression e specifies a mapping (i.e., the computational field) associating each computation round of each device to the value that e assumes at that space-time event. This duality intrinsically supports the reconciliation between the local behaviour of each device and the emerging global behaviour of the whole network of devices [69, 66], as proved by the computational adequacy and abstraction properties in [38], which relate operational and denotational semantics.

Figure 3 gives an abstract syntax for field calculus, as presented in recent works [70]. In this syntax, the overbar notation $\overline{\texttt{e}}$ indicates a sequence of elements (i.e., $\overline{\texttt{e}}$ stands for $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$), and multiple overbars are expanded together (e.g., $\overline{\delta} \mapsto \overline{\ell}$ stands for $\delta_1 \mapsto \ell_1, \delta_2 \mapsto \ell_2, \ldots, \delta_n \mapsto \ell_n$ which is a map associating local values to device identifiers). There are four keywords in this syntax: def for function definition; if for (the field-based variation of) branching expression; and rep and nbr for the two peculiar constructs of field calculus, respectively responsible for evolution of state over time and for sharing information between neighbours.

A field calculus program P consists of a sequence of function declarations $\overline{\texttt{F}}$ followed by the main expression e, defining global (and also local) behaviour of the aggregate system. An expression e can be:

- A *variable* x, e.g., a function parameter.

- A *value* v, which can be of the following two kinds:

    - a *local value* ℓ, defined via data constructor c and arguments $\overline{\ell}$, such as a Boolean, number, string, pair, tuple, etc;

    - a *neighbouring (field) value* ϕ representing a collection of values from nearby devices, in the form of a function that associates, for each device, the set of neighbour devices δ (including the device itself) to local values ℓ, e.g., a map of neighbours to the distances to those neighbours.

- A function call $\texttt{f}(\overline{\texttt{e}})$ to either a *user-declared function* d (declared with the def keyword) or a *built-in function* b, such as a mathematical or logical

operator, a data structure operation, or a function returning the value of a sensor.

- A *branching expression* $\mathtt{if}(\mathtt{e_1})\{\mathtt{e_2}\}\{\mathtt{e_3}\}$, used to split a computation into isolated sub-regions where (and when) $\mathtt{e_1}$ evaluates to $\mathtt{True}$ or $\mathtt{False}$: the result is computation of $\mathtt{e_2}$ in the former area, and $\mathtt{e_3}$ in the latter.

- The $\mathtt{nbr}\{\mathtt{e}\}$ construct, which creates a neighbouring value mapping neighbours to their latest available result of evaluating $\mathtt{e}$. In particular, each device δ:
    1. shares its value of $\mathtt{e}$ with its neighbours, and
    2. evaluates the expression into a neighbouring value ϕ, where ϕ is a function that maps each neighbour δ' of δ to the latest evaluation of $\mathtt{e}$ that has been shared from δ'.

    For instance, $\mathtt{nbr}\{\mathtt{temperature}()\}$ (where $\mathtt{temperature}$ is a built-in sensor estimating local temperature) would produce a neighbouring value ϕ associating to each neighbour the temperature measured by that neighbour. Note that in an $\mathtt{if}$ branch, sharing is restricted to occur between devices within the same subspace of the branch (since devices in a different subspace do not execute the same $\mathtt{nbr}\{\mathtt{e}\}$ constructs).

- The $\mathtt{rep}(\mathtt{e_1})\{(\mathtt{x}){=}{>}\mathtt{e_2}\}$ construct, which models state evolution over time. This construct retrieves the value $\mathtt{v}$ computed for the whole $\mathtt{rep}$ expression in the last evaluation round (the value produced by evaluating the expression $\mathtt{e_1}$ is used at the first evaluation round) and updates it with the value produced by evaluating the expression obtained from $\mathtt{e_2}$ by replacing the occurrences of $\mathtt{x}$ by $\mathtt{v}$.

Within this collection of operations, the $\mathtt{nbr}$ and $\mathtt{rep}$ constructs are special, handling message exchanges respectively between devices and within rounds of a single device. These constructs are assumed to be backed by a data gathering mechanism accomplished through a process called *alignment* [71], which ensures appropriate message matching, i.e., that no two different instances of a $\mathtt{nbr}$ expression can inadvertently "swap" their respective messages, nor can two different instances of a $\mathtt{rep}$ expression "swap" their state memory. This has the notable consequence that the two branches of an $\mathtt{if}$ statement in field calculus are executed in isolation: a device computing the "then" branch cannot communicate with the "else" branch of a neighbour, and vice versa.

Example 3.1.1 (Distance Avoiding Obstacles). Consider Figure 4. Function $\mathtt{distanceTo}$ takes as argument a field of Booleans $\mathtt{source}$, associating true to *source nodes*, and produces as result a field of reals, mapping each device to its minimum distance to a source node, as computed by relaxation of the triangle inequality; namely: repetitively, and starting from infinity (construct $\mathtt{rep}$) everywhere, the distance on any node gets updated to 0 on source nodes (function $\mathtt{mux}(\mathtt{c},\mathtt{t},\mathtt{e})$ is a purely functional multiplexer which chooses $\mathtt{t}$ if $\mathtt{c}$ is $\mathtt{true}$, or $\mathtt{e}$ otherwise), and elsewhere to the minimum (built-in $\mathtt{minHood}$) of neighbours'

```
// distance from source region with nbrRange metric
def distanceTo(source) {
  rep (Infinity) { (dist) =>
     mux ( source, 0, minHood(nbr{dist} + nbrRange()) )
  }
}
// distance from source region, avoiding obstacle region
def distanceToWithObs(source, obstacle) {
  if (obstacle) { Infinity }{ distanceTo(source) }
}
// main expression
distanceToWithObs(deviceId() == 0, senseObs())
```

Figure 4: Example field calculus code

distance (construct `nbr`) added with `nbrRange`, a sensor for estimated distances. Function `distanceToWithObs` takes an additional argument, a field of booleans `obstacle`, associating true to *obstacle nodes*; it partitions the space of devices: on obstacle nodes it gives the field of infinity values, elsewhere it uses computation of `distanceTo`. Because of alignment, the set of neighbours considered for `distanceTo` automatically discards nodes that evaluate the other branch of `if`, effectively making computation of distances circumvent obstacles. Finally, the main expression calls `distanceToWithObs` to compute distances from the node with `deviceId` equal to `0`, circumventing the devices where `senseObs` gives true.

Example 3.1.2 (Monitor). Consider the following field calculus expression.

```
if ( fail() ) { rep (0) {(x) => x-1} } { sumHood(nbr{1}) }
```

This expression represents a simple monitor, for which higher values indicate a good situation, while lower (negative) values signal problematic situations. In devices where `fail` is true, the number of consecutive rounds of failure is counted with negative numbers by the `rep` expression. Non-failing devices instead compute `sumHood(nbr1)` (isolated from failing devices) which *(i)* builds a neighbouring field ϕ mapping each non-failing neighbour to 1; *(ii)* sums every value in the range of ϕ (except that for the current device) with built-in `sumHood`, obtaining the (non-negative) total number of non-failing neighbours.

3.2. Operational semantics, typing and basic properties

The distinguished interaction model of this approach has been first formalised in [66] (see also [69]) by means of a small-step operational semantics modelling single device computation (which is ultimately responsible for the whole network execution). The main technical novelty in this formalisation is that device state and message content are represented in an unified way as an annotated evaluation tree θ. Field construction, propagation, and restriction are then supported by local evaluation "against" the collection Θ of evaluation trees received from neighbours. The alignment mechanism to ensure appropriate message matching is then implemented by operations navigating these

trees, and discarding them whenever different branches are taken (to prevent unwanted communication between `nbr` constructs in different branches of an `if` expression).

Recent work models single device computation by a big-step operational semantics [70], expressed by the judgement $\delta; \Theta; \sigma \vdash \mathtt{e_{main}} \Downarrow \theta$, to be read "expression $\mathtt{e_{main}}$ evaluates to θ on device δ with respect to environment Θ and sensor state σ". The overall network evolution is then formalised by a small-step operational semantics as a transition system $N \xrightarrow{act} N$ on network configurations N, in which actions act can either be environment changes or single device computations (in turn modelled by the big-step semantics). For the purpose of this survey, it is key to convey the overall behaviour of a field computation (modelled by the denotational semantics formalised in the next section), but there is no need to provide the full details of the operational semantics, since they pertain to a specific implementation "template" for node computational rounds—the interested reader may find it in [70, Online Appendix C].

The work in [69] presents a type system used to intercept ill-formed field-calculus programs. Figure 5 presents this system (adapted to the syntax in Figure 3), which builds on the Hindley-Milner type system [72] for ML-like functional languages, as a set of syntax-directed type inference rules. Being syntax-directed, the rules straightforwardly specify a variant of the Hindley-Milner type inference algorithm [72]. Namely, an algorithm that, given a field calculus expression and type assumptions for its free variables, either fails (if the expression cannot be typed under the given type assumptions) or returns its principal type, i.e., a type such that all the types that can be assigned to an expression by the type inference rules can be obtained from the principal type by substituting type variables with types.

Types are partitioned in two sets: *types for expressions T* and *type schemes for functions FS* (constructors, built-in operators and user-defined functions)—this reflects the fact that the base field calculus does not support higher order functions (i.e., functions are not values). Expression types are further partitioned in two sets:

- types for *local values L*, including Booleans and other built-in types such as numbers, strings, pairs, tuples, etc;

- types for *neighbouring values F*, e.g., the values produced by `nbr`-expressions.

These sets also include two kinds of type variables t and l (similar to how the Standard ML type system features equality and non-equality type variables [73]). This allows functions to behave polymorphically while enforcing ad-hoc restrictions necessary to guarantee the properties discussed at the end of this section.

Type environments, ranged over by $\mathcal{A}$, collect type assumptions $\mathtt{x} : T$ for program variables. *Function-type-scheme environments*, ranged over by $\mathcal{D}$, collect the function type schemes $\mathtt{f} : \forall \bar{t}\bar{l}.(\overline{T}) \rightarrow T$ for the data constructors and

Types:

$$
\begin{array}{llll}
T & ::= & t \mid L \mid F & \text{expression type} \\
L & ::= & l \mid \texttt{bool} \mid \ldots & \text{local type} \\
F & ::= & \texttt{field}(L) & \text{field type}
\end{array}
$$

Function type schemes:

$$
\begin{array}{llll}
FS & ::= & \forall \overline{t}\overline{l}.(\overline{T}) \to T & \text{function type scheme}
\end{array}
$$

Expression typing: $\boxed{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}$

$$
\text{[T-VAR]} \quad \frac{}{\mathcal{D}; \mathcal{A}, \texttt{x} : T \vdash \texttt{x} : T}
\qquad
\text{[T-DAT]} \quad \frac{\mathcal{D}_0; \emptyset \vdash \overline{\ell} : \overline{L} \qquad \texttt{c} : \forall \overline{l}.(\overline{L}') \to L' \; \in \; \mathcal{D}_0 \qquad \overline{L} = \overline{L}'[\overline{l} := \overline{L}''] \quad L = L'[\overline{l} := \overline{L}'']}{\mathcal{D}; \mathcal{A} \vdash \texttt{c}(\overline{\ell}) : L}
$$

$$
\text{[T-APP]} \quad \frac{\mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T} \qquad \texttt{f} : \forall \overline{t}\overline{l}.(\overline{T}') \to T' \; \in \; \mathcal{D} \qquad \overline{T} = \overline{T}'[\overline{t} := \overline{T}'', \; \overline{l} := \overline{L}''] \quad T = T'[\overline{t} := \overline{T}'', \; \overline{l} := \overline{L}'']}{\mathcal{D}; \mathcal{A} \vdash \texttt{f}(\overline{\texttt{e}}) : T}
$$

$$
\text{[T-IF]} \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_c : \texttt{bool} \qquad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_T : L \qquad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_F : L}{\mathcal{D}; \mathcal{A} \vdash \texttt{if}(\texttt{e}_c)\{\texttt{e}_T\}\{\texttt{e}_F\} : L}
$$

$$
\text{[T-REP]} \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : L \qquad \mathcal{D}; \mathcal{A}, \texttt{x} : L \vdash \texttt{e}_2 : L}{\mathcal{D}; \mathcal{A} \vdash \texttt{rep}(\texttt{e}_1)\{(\texttt{x}){=}{>}\texttt{e}_2\} : L}
\qquad
\text{[T-NBR]} \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : L}{\mathcal{D}; \mathcal{A} \vdash \texttt{nbr}\{\texttt{e}\} : \texttt{field}(L)}
$$

Function typing: $\boxed{\mathcal{D} \vdash \texttt{F} : FS}$

$$
\text{[T-FUNCTION]} \quad \frac{\mathcal{D}, \texttt{d} : \forall \overline{t}\overline{l}.(\overline{T}) \to T; \; \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : T}{\mathcal{D} \vdash \texttt{def } \texttt{d}(\overline{\texttt{x}}) \, \{\texttt{e}\} : \forall \overline{t}\overline{l}.(\overline{T}) \to T}
$$

Program typing: $\boxed{\mathcal{D}_0 \vdash \texttt{P} : T}$

$$
\text{[T-PROGRAM]}
$$
$$
\frac{\texttt{F}_i = (\texttt{def } \texttt{d}_i(\_) \, \_) \qquad \mathcal{D}_{i-1} \vdash \texttt{F}_i : FS_i \qquad \mathcal{D}_i = \mathcal{D}_{i-1}, \texttt{d}_i : FS_i \qquad (i \in 1..n) \qquad \mathcal{D}_n; \emptyset \vdash \texttt{e} : T}{\mathcal{D}_0 \vdash \texttt{F}_1 \cdots \texttt{F}_n \; \texttt{e} : T}
$$

Figure 5: Hindley-Milner typing for field calculus, as adapted from [69]

built-in functions together with the function type schemes inferred for the user-defined functions. In particular, the distinguished *built-in function-type-scheme environment* $\mathcal{D}_0$ associates a function type scheme to each data constructor $\texttt{c}$ and built-in function $\texttt{b}$.

The typing judgement for expressions is of the form "$\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T$", to be read: "$\texttt{e}$ has type T under the function-type-scheme assumptions $\mathcal{D}$ and the type assumptions $\mathcal{A}$". The typing judgement for function declarations "$\mathcal{D} \vdash \texttt{F} : FS$" and programs "$\mathcal{D}_0 \vdash \texttt{P} : T$" are read analogously. We say that a program $\texttt{P}$ is *well-typed* to mean that $\mathcal{D}_0 \vdash \texttt{P} : T$ holds for some type T.

The typing rules are given in Figure 5 (bottom). Rules [T-VAR], [T-DAT], [T-APP], [T-FUNCTION], and [T-PROGRAM] are almost standard. Rule [T-NBR] requires the argument of an `nbr` construct to be local, in order to prevent the creation of a *field of fields*, which would be computationally expensive. Rules [T-IF] and [T-REP] require the type of their arguments to match and to be local, since `rep` or `if` constructs of field type would violate the *domain alignment* property. In fact, this type system is proved to guarantee the following two valuable properties for field calculus:

- *Domain alignment:* On each device, the domain of every neighbouring value arising during the reduction of a well-typed expression consists of the identifiers of the aligned neighbours and of the identifier of the device itself. In other words, information sharing is scoped to precisely implement the aggregate abstraction.

- *Type soundness:* The reduction of a well-typed expression does not get stuck.

Example 3.2.1 (Typing). Consider the Examples 3.1.1 and 3.1.2. The type system assigns the following types to the involved built-in functions, user-defined functions, and main expressions.

```
// minHood, sumHood : (bool) -> num
// nbrRange : () -> field(num)
def distanceTo(source) ... // (bool) -> num
def distanceToWithObs(source, obstacle) ... // (bool, bool) -> num
distanceToWithObs(deviceId() == 0, senseObs()) // num
if ( fail() ) { rep (0) {(x) => x-1} } { sumHood(nbr{1}) } // num
```

3.3. Denotational semantics

The operational semantics corresponds to the *local* interpretation of the field calculus: it specifies details concerning how a device internally processes a round, what information fills the message sent to neighbours, and which information persists on a node across time. Such a specification is, of course, abstract, as a compliant implementation can apply optimisation techniques (size of messages and of state information) that need not be specified into an operational semantics. A further, more abstract formalisation of field calculus can be given by a denotational semantics focussing on the *global* interpretation of field expressions, namely, as functions from (space-time) fields to fields. As advocated in [38], this allows one to focus on the semantics of field constructs in a way that completely abstracts from local interpretation of expressions, i.e., considers only their global effect. Ideally, this is the semantics one has in mind when designing complex specifications, whereas operational semantics is more a concern of designers of field calculus support (interpreters, platforms). Accordingly, and as a novel contribution of this paper, in this section we present a denotational semantics for the field calculus (summarised in Figure 7), obtained by adapting the denotational semantics of the higher-order field calculus (Section 3.5) given
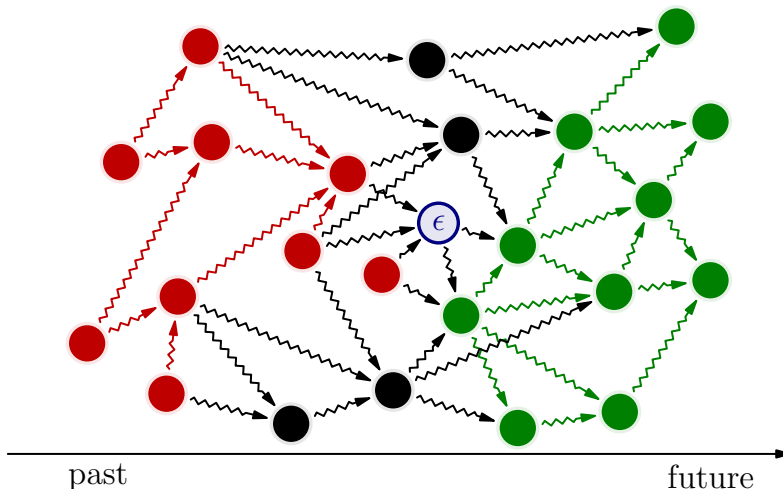
Figure 6: Example of a space-time event structure, comprising events (circles) and neighbour relations (arrows). Colours indicate causal structure with respect to event ϵ, splitting events into causal past (red), causal future (green), and concurrent (non-ordered, in black). Figure adapted from [74].

in [38]. The resulting first-order version is much simpler[4] and therefore more suitable for a survey paper.

In this semantics, each round of computation happening on a device is represented by an *event* ϵ, and the collection of all such executions across space (i.e., across devices) and time (i.e., over multiple rounds) forms an *event structure* **E** [75], representing the overall execution of a single aggregate machine. Note that we rely on a true-concurrent semantics, which is more faithful to the intended real-world applications of the field calculus. Each event structure is assumed to be equipped with a *neighbouring* relation $\leadsto$ guided by message exchanges, so that $\epsilon_1 \leadsto \epsilon_2$ iff a message sent in ϵ_1 was taken into account in ϵ_2. This relation provides a topology for **E**, and its transitive closure forms the irreflexive *causality* partial order $<$[5]. Figure 6 shows an example of such an event structure, showing how these relations partition events into "causal past", "causal future", and non-ordered "concurrent" subspaces with respect to any given event ϵ. Interpreting this in terms of physical devices and message passing, a physical device is instantiated as a chain of events connected by $\leadsto$ relations (representing evolution of state over time with the device carrying state from one event to the next), and any $\leadsto$ relation between devices represents informa-

---

[4]In particular, the denotation of function types is greatly simplified by omitting name tags and referring to the local behaviour instead of the global one, rules for denoting functions are not needed and the denotation of function applications is simplified by substituting nested limits with a single global limit on a stack trace length parameter.

[5]We require the past of each event $\{\epsilon' \in \mathbf{E} : \epsilon' < \epsilon\}$ to be finite (i.e., every computation has a start), even though the overall set of events may be infinite (eternal computations).

tion exchange from the tail neighbour to the head neighbour. Notice that this is a very flexible and permissive model: there are no assumptions about synchronisation, shared identifiers or clocks, or even regularity of events (though of course these things are not prohibited either).

We assume that each ϵ incorporates all the relevant information about the corresponding event, e.g., the involved device δ_ϵ and its sensor state at the time the event happened. For all ϵ and δ, we define ϵ^δ as the latest event at δ that ϵ can be aware of, namely the one[6] satisfying $\delta = \delta_{\epsilon^\delta}$ and $\epsilon^\delta \rightsquigarrow \epsilon$ if $\delta \neq \delta_\epsilon$, or $\epsilon^\delta = \epsilon$ in case $\delta = \delta_\epsilon$. We also define $E^-(\epsilon)$ where $\epsilon \in E \subseteq \mathbf{E}$ as the set of devices δ such that ϵ^δ exists in E. Finally, we use ϵ^- to denote the previous event of ϵ at the same device *if it exists*, and E_0 to denote the set of initial events ($\epsilon \in E$ such that ϵ^- does not exist).

In the remainder of this section, we use the convention that a partial function $\lambda x \in X.\phi(x)$ is defined only on the elements x for which each subformula of $\phi(x)$ is defined, and thus propagate undefinedness implicitly.

Figure 7 (top frame) presents the interpretation $\mathcal{T}[\![T]\!]$ of types and $\mathcal{T}[\![FS]\!]$ of function type schemes, as the set of possible values. We assume that the interpretation of local types is given, and define the interpretation of field types $\mathtt{field}(L)$ as partial functions[7] from device identifiers $\mathbf{D}$ to the corresponding local type interpretation $\mathcal{T}[\![L]\!]$. Function type schemes are interpreted as functions between the corresponding interpretations, after applying any of the allowed substitutions of the type variables $\overline{tl}$.

The interpretation of type schemes is used to define the interpretation of built-in functions and constructors (middle frame), through the interpretation function $\mathcal{B}[\![\cdot]\!]$, which we assume to be given. The main objects of the denotational semantics are *space-time values*, which are partial maps from a given set of events $\mathbf{E}$ (implicitly equipped with a neighbouring relation $\rightsquigarrow$) to values taken from the interpretation of the corresponding type. The interpretation of expressions $\mathcal{E}[\![\mathtt{e}]\!]_X^E$ produces a space-time value, and is performed with respect to a subset $E \subseteq \mathbf{E}$ and to a variable environment X, which is a map from variable names (we use $\mathtt{X}$ for the set of possible variable names) to space-time values of the corresponding type.

Figure 7 (bottom frame) defines the interpretation of expressions, as a limit on a parameter n that is to be understood as a maximum allowed recursive depth. The interpretation of variables directly exploits the parameter X. The interpretation of the application of built-in functions and constructors is directly delegated to the built-in interpretation function $\mathcal{B}[\![\cdot]\!]$. The interpretation of the application of defined functions returns an empty space-time value when the recursive depth is exhausted ($n = 0$). Given a positive recursive depth $n + 1$, it instead corresponds to the interpretation of the body with a reduced recursion depth n, using additional assumptions for the function parameters[8] (calculated

---

[6]We require that $\mathbf{E}$ is such that ϵ^δ is always unique (if it exists).

[7]We write the set of partial functions from A to B as $A \rightharpoonup B$.

[8]We assume that the arguments of defined functions $\mathtt{d}$ are implicitly renamed to avoid

Denotation of types and type schemes: $\boxed{\mathcal{T}[\![T]\!], \mathcal{T}[\![FS]\!]}$

$$\mathcal{T}[\![L]\!] \quad \text{given}$$
$$\mathcal{T}[\![\texttt{field}(L)]\!] = \mathbf{D} \twoheadrightarrow \mathcal{T}[\![L]\!]$$
$$\mathcal{T}[\![\forall \bullet .(\overline{T}) \to T]\!] = (\mathcal{T}[\![\overline{T}]\!]) \to \mathcal{T}[\![T]\!]$$
$$\mathcal{T}[\![\forall \overline{t}\overline{l}.(\overline{T}) \to T]\!] = \bigcup_{\overline{T'}\,\overline{L'}} \mathcal{T}[\![\forall \bullet .((\overline{T}) \to T)[\overline{t} := \overline{T}', \overline{l} := \overline{L}']]\!]$$

Interpretation functions:

$\mathcal{B}[\![\texttt{b}]\!] \in \mathcal{T}[\![FS]\!]$	where $\texttt{b} : FS$	built-in interpretation
$\mathcal{B}[\![\texttt{c}]\!] \in \mathcal{T}[\![FS]\!]$	where $\texttt{c} : FS$	constructors interpretation
$\mathcal{V}[\![T]\!] = \mathbf{E} \twoheadrightarrow \mathcal{T}[\![T]\!]$		space-time values
$\mathcal{E}[\![\texttt{e}]\!]_X^E \in \mathcal{V}[\![T]\!]$	where $\texttt{e} : T$ under $\overline{\texttt{x}} : \overline{T}$	expression interpretation
$X \in \texttt{X} \to \bigcup_T \mathcal{V}[\![T]\!]$		variable environment

---

Denotation of expressions: $\boxed{\mathcal{E}[\![\texttt{e}]\!]_X^E}$

$$\mathcal{E}[\![\texttt{e}]\!]_X^E = \lim_n \mathcal{E}[\![\texttt{e}]\!]_X^{E,n}$$

$$\mathcal{E}[\![\texttt{x}]\!]_X^{E,n} = \lambda \epsilon \in E.X(\texttt{x})(\epsilon)$$

$$\mathcal{E}[\![\texttt{c}(\overline{\ell})]\!]_X^{E,n} = \lambda \epsilon \in E.\mathcal{B}[\![\texttt{c}]\!](\mathcal{E}[\![\overline{\ell}]\!]_X^{E,n}(\epsilon))$$

$$\mathcal{E}[\![\texttt{b}(\overline{\texttt{e}})]\!]_X^{E,n} = \lambda \epsilon \in E.\mathcal{B}[\![\texttt{b}]\!](\mathcal{E}[\![\overline{\texttt{e}}]\!]_X^{E,n}(\epsilon))$$

$$\mathcal{E}[\![\texttt{d}(\overline{\texttt{e}})]\!]_X^{E,0} = \emptyset$$
$$\mathcal{E}[\![\texttt{d}(\overline{\texttt{e}})]\!]_X^{E,n+1} = \lambda \epsilon \in E.\mathcal{E}[\![body(\texttt{d})]\!]_{X \cup args(\texttt{d}) \mapsto \mathcal{E}[\![\overline{\texttt{e}}]\!]_X^{E,n+1}}^{E,n}$$

$$\mathcal{E}[\![\texttt{if}(\texttt{e}_c)\{\texttt{e}_T\}\{\texttt{e}_F\}]\!]_X^{E,n} = \mathcal{E}[\![\texttt{e}_T]\!]_X^{E_{\text{True}},n} \cup \mathcal{E}[\![\texttt{e}_F]\!]_X^{E_{\text{False}},n}$$
$$\text{where } E_\ell = \{\epsilon \in E : \ \mathcal{E}[\![\texttt{e}_c]\!]_X^{E,n}(\epsilon) = \mathcal{B}[\![\ell]\!]()\}$$

$$\mathcal{E}[\![\texttt{nbr}\{\texttt{e}\}]\!]_X^{E,n} = \lambda \epsilon \in E.\lambda \delta \in \overline{E}(\epsilon).\mathcal{E}[\![\texttt{e}]\!]_X^{E,n}(\epsilon^\delta)$$

$$\mathcal{E}[\![\texttt{e}]\!]_X^{E,n} = \lim_k \mathcal{R}^k[\![\texttt{e}]\!]_X^{E,n} \text{ where } \texttt{e} = \texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{ => } \texttt{e}_2\} \text{ and}$$
$$\mathcal{R}^0[\![\texttt{e}]\!]_X^{E,n} = \emptyset$$
$$\mathcal{R}^{k+1}[\![\texttt{e}]\!]_X^{E,n} = \mathcal{E}[\![\texttt{e}_2]\!]_{X \cup \texttt{x} \mapsto \mathbf{shift}(\mathcal{R}^k[\![\texttt{e}]\!]_X^{E,n}, \mathcal{E}[\![\texttt{e}_1]\!]_X^{E,n}(\epsilon))}^{E,n}$$

Figure 7: Denotational semantics of the field calculus

at the same recursive depth $n + 1$).

The interpretation of branching statements is produced by adjoining the interpretation of its two branches, each performed within the sole events on which the condition produced the corresponding (true or false) result. The interpretation of $\texttt{nbr}\{\texttt{e}\}$ expressions produces in each event $\epsilon \in E$ a neighbouring field value associating to each $\delta \in \overline{E}(\epsilon)$ the interpretation of $\texttt{e}$ in ϵ^δ. Finally, the interpretation of $\texttt{e} = \texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{ => } \texttt{e}_2\}$ statements is defined as a limit of partial

---

clashes with existing variables in X whenever needed.

interpretations $\Phi_k = \mathcal{R}^k [\![\mathtt{e}]\!]_X^{E,n}$, defined for events with at most k predecessors on the same device. When $k = 0$, the interpretation is an empty space-time value. For positive values $k + 1$, the interpretation of $\mathtt{e}$ corresponds to the interpretation of $\mathtt{e}_2$ with the additional assumption that $\mathtt{x}$ is $\mathbf{shift}(\Phi_k, \Phi')$, where Φ' is the interpretation of $\mathtt{e}_1$ and

$$\mathbf{shift}(\Phi, \Phi') = \lambda \epsilon \in E. \begin{cases} \Phi'(\epsilon) & \epsilon \in E_0 \\ \Phi(\epsilon^-) & \text{otherwise} \end{cases}$$

"pushes" each value in Φ to the next future event, while falling back to Φ' for initial events.

We remark that the field calculus, according to the given semantics, is *Turing universal* for distributed computations as shown in [74]. More precisely, field calculus programs can simulate the behaviour of any Turing machine TM_{cone} that receives in each event ϵ the collection of all data available in each past event $\{\epsilon' \in \mathbf{E} : \epsilon' < \epsilon\}$ and correspondingly produces an output value.

Example 3.3.1 (Monitor Semantics). In order to showcase the denotational semantics at work, consider the field calculus expression of Example 3.1.2.

```
if ( fail() ) { rep (0) {(x) => x-1} } { sumHood(nbr{1}) }
```

The denotational semantics of this expression is summarised in Figure 8, on a sample event structure E consisting of 21 events occurring on 4 devices, among which device 2 had a reboot after its first two rounds of failure (represented by the missing link between consecutive events). The variable environment is initially empty $X = \emptyset$ and the recursive depth n is irrelevant as no function calls are considered. The denotation of the whole expression $\mathtt{e}$ is first split into two sub-networks $E_{\mathtt{True}}, E_{\mathtt{False}}$ depending on the value returned by the built-in function $\mathtt{fail}$.

The denotation of the *then* branch $\mathtt{e}_r$ is obtained as a limit of partial approximations: the first $\mathcal{R}^0[\![\mathtt{e}_r]\!]_\emptyset$ is defined only on initial events (as the result of $\mathtt{x} - 1$ assuming that $\mathtt{x}$ is 0 on those events), and the following ones are defined on more events until the limit is reached with $\mathcal{R}^3[\![\mathtt{e}_r]\!]$.

The denotation of the *else* branch $\mathtt{e}_s$ is obtained in steps. First, the denotation of 1 (top right) is used to compute the denotation of $\mathtt{nbr}\{1\}$ (middle right), producing in each event a neighbouring field value ϕ associating 1 (shown as edge label) to each neighbour device. Finally, such neighbouring field values are summed up by the built-in function $\mathtt{sumHood}$ (bottom right).

3.4. Behavioural properties

The field calculus is designed as a general-purpose language for spatially distributed computations. Thus, regularity properties have been isolated and studied for subsets of the core language. Among them, the established notion of *self-stabilisation* to correct states for distributed systems [76, 77, 78] plays a central role. This notion, defined in terms of properties of the transition system
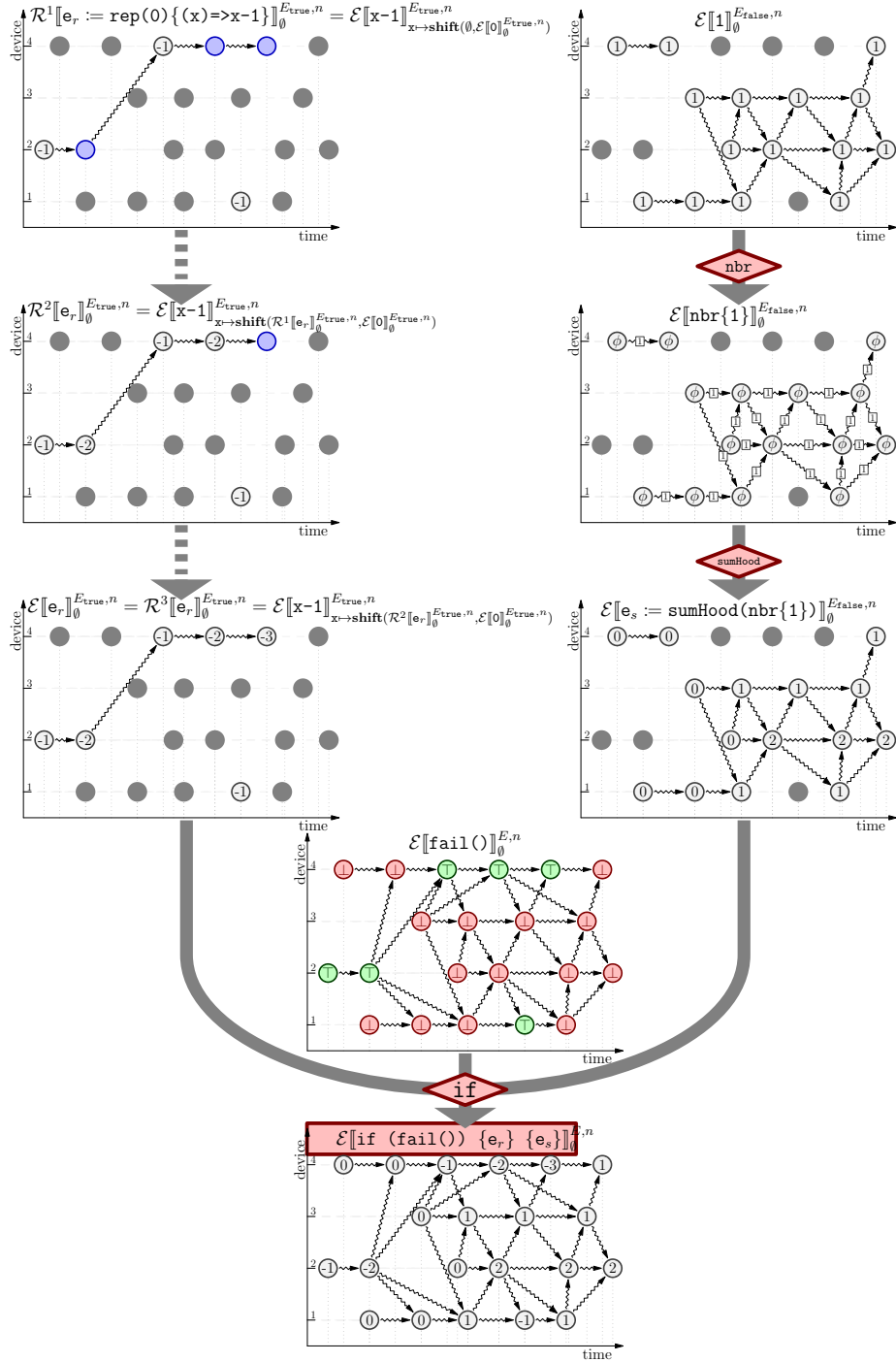
Figure 8: Example of denotational semantics of an expression as it is built up from sub-expressions. Events not included in the current reference event structure are greyed out, while events on which the denotation is undefined are marked in blue.

$N \xrightarrow{act} N$ of network evolution (cf. Section 3.2), ensures that both *(i)* the evaluation of a program on an eventually constant input converges to a limit value in each device in finite time; *(ii)* this limit only depends on the input values, and not on the transitory input values that may have happened before that. When applied in a dynamically evolving system, a self-stabilising algorithm guarantees that whenever the input changes, the output reacts accordingly without spurious influences from past values.

In [79] (an extended version of [80]), a first self-stabilising fragment is isolated through a *spreading* operator, which minimises neighbour values as they are monotonically updated by a *diffusion* function. This pattern can be composed arbitrarily with local operations, but no explicit `rep` and `nbr` expressions are allowed: nonetheless, several building blocks can be expressed inside this fragment, such as classic distance estimation and broadcast (specific instances of operator G in Figure 12).

More self-stabilising programs and existing "building block" implementations are covered by the larger self-stabilising fragment introduced in [70] (an extended version of [81]). This fragment restricts the usage of `rep` statements to three specific patterns: converging, acyclic, and minimising `rep`. They roughly correspond to the three main building blocks proposed, G, C and T: G is a generalisation of distance estimation, which spreads a spanning tree from a source region based on a given metric, and use it to compute values outward; C conversely collects values inward a spanning tree (typically produced by G) aggregating them "en route" so as to summarise a final result into a target node; and finally T is a local operator to temporally evolve a value until reaching a fixpoint—see Figure 12). Furthermore, a notion of *equivalence* and *substitutability* for self-stabilising programs is examined: on the one hand, this notion allows for practical optimisation of distributed programs by substitution of routines with equivalent but better-performing alternatives; on the other hand, this equivalence relation naturally induces a *limit* viewpoint for self-stabilising programs, complementing and integrating the two general (local and global) viewpoints by abstracting away the transitory characteristics and isolating the input-output mapping corresponding to the distributed algorithm. These viewpoints effectively constitute different semantic interpretations of the same program: operational semantics (local viewpoint), denotational semantics (global viewpoint), and eventual behaviour (limit viewpoint).

A fourth "continuous" viewpoint is considered in [82]: as the density of computing devices in a given area increases, assuming that each device takes inputs from a single continuous function on a space-time manifold, the output values may converge towards a limit continuous output. Programs with this property are called *consistent*, and have a "continuous" semantic interpretation as a transformation of continuous functions on space-time manifolds. Taking inspiration from self-stabilisation, this notion is relaxed for *eventually consistent* programs, which are only required to continuously converge to a limit except for a transitory initial period, *provided* that the inputs are constant (except for a transitory initial period). Eventual consistency can then be proved for

all programs expressible in the GPI (gradient-following path integral) calculus, which is a restriction of the field calculus where the only coordination mechanism allowed is the GPI operator, a generalised variant of the distance estimation building block.

Finally, a recent thread of work [83] has begun considering the transient behaviour of field calculus programs, by providing real-time guarantees on program performance. In these results, a bounded amount of error with respect to ideal values is proved to hold after a predictable set-up (or reconfiguration) time.

Up to this point, hence, validation of behavioural properties is mostly addressed "by construction", namely, proving properties on simple building blocks or restricting the calculus to fragments. It is a future work to consider the applicability of techniques such as the formal basis in [77], or model-based analysis such as [84].

3.5. Language extension: the higher-order field calculus

The higher-order field calculus (HFC) [38, 85] is an extension of the field calculus with first-class functions. Its primary goal is to allow programmers to handle functions just like any other value, so that code can be dynamically injected, moved, and executed in network (sub)domains. Namely, in HFC:

- Functions can take functions as arguments and return a function as result (higher-order functions). This is key to define highly reusable building block functions, which can then be fully parametrised with various functional strategies.

- Functions can be created "on the fly" (anonymous functions). Among other applications, such functions can be passed into a system from the external environment, as a field of functions considered as input coming from a sensor modelling addition of new code into a device while the system is operating.

- Functions can be moved between devices (via the `nbr` construct) and the function to be executed can be remembered and changed over time (via the `rep` construct), which allows one to express complex patterns of code deployment across space and time.

- A field of functions (possibly created on the fly and then shared by movement to all devices) can be used as an "aggregate function" operating over a whole spatial domain.

In considering fields of function values, HFC takes an approach in which making a function call acts as a branch, with each function in the range of the field applied only on the subspace of devices that hold that function. When the field of functions is constant, this implicit branch reduces to be precisely equivalent to a standard function call. This means that we can view ordinary evaluation of a function name (or anonymous function) as equivalent to creating a function-valued field with a constant value, then making a function call applying that

field to its argument fields. This elegant transformation is one of the key insights of HFC, enabling first-class functions to be implemented with relatively minimal complexity.

In [85] the operational semantics of HFC is formalised, for computation within a single device, by a big-step operational semantics where each expression evaluates to an ordered tree of values tracking the results of all evaluated sub-expressions. Moreover, [85] also presents a formalisation of network evolution, by a transition system on network configurations—transitions can either be firings of a device or network configuration changes, while network configurations model environmental conditions (i.e., network topology and inputs of sensors on each device) and the overall status of devices in the network at a given time. In the extension of this work in [38] the formalisation of HFC is carried on by providing a denotational semantics, which is proved to correspond to the operational semantics through *computational adequacy and abstraction* results. Furthermore, a refined type system is presented that is able to guarantee *domain alignment*, i.e., that the domain of any expression of field type equals the set of neighbours that computed the same expression.

4. From Field Calculus to Aggregate Computing

In this section, we discuss the current state of the art in practical aggregate computing, without going into deep technical details—the reader can access code examples and tutorials from the references provided. We begin by discussing the construction of implementations of field calculus as supported by the domain specific language Protelis (Section 4.1) and the SCAFI API for Scala (Section 4.2). We then discuss the layered abstractions of aggregate programming built upon these foundations, from resilient operators to pragmatic libraries (Section 4.3). Note that as far as current implementations are concerned, field calculus is supported in its higher-order version, hence in the following we sometimes generally refer to field calculus even if higher-order capabilities are concerned.

4.1. Protelis: a DSL for field calculus

The concrete usage of field calculus in application development is dependent on the availability of practical languages, which provide an interpreter or compiler, as well as handling runtime aspects such as communication, interfacing with the operating system, and integration with existing software. Protelis [86] provides one such implementation, including: (i) a concrete syntax; (ii) an interpreter and a virtual machine; (iii) a device interface abstraction and API; and (iv) a communication interface abstraction and API.

In Protelis, the parser translates a Protelis source code file into a valid representation of HFC semantics. This translated program, along with an execution context, is fed to a virtual machine that executes the Protelis interpreter at regular intervals. The execution context API defines the interface towards the operating system, including (with ancillary APIs) an abstraction of the device's capabilities and communication system. This architecture has been

23

demonstrated to make the language easy to port across diverse contexts, both simulated (Alchemist[9] [87] and NASA World Wind [88]) and real-world [89].

The entire Protelis infrastructure is developed in Java and hosted on the Java Virtual Machine (JVM). The motivation behind this choice is twofold: first, the JVM is highly portable, being available on a variety of architectures and operating systems; second, the Java world is rich in libraries that can be directly used within Protelis, with little or no need for writing new libraries for common tasks.

The model-to-model translation between the Protelis syntax and the HFC interpreter is implemented using the Xtext framework [90]. Along with the parser machinery, this framework is also able to generate most of the code required for implementing Eclipse plug-ins: one such plug-in is available for Protelis, assisting the developer through code highlighting, completion suggestions, and early error detection.

The language syntax is designed with the goal of lowering the learning curve for the majority of developers, and as such it is inspired by languages of the C-family (C, C++, Java, C#, ...), with some details borrowed from Python. Code can be organised in modules (or namespaces) whose name must reflect the directory structure and the file name. Modules can contain functions and a main script. The code snippet in Figure 9 offers a sampler of both the ordinary and field-calculus-specific features of Protelis, including importing libraries and static methods, using functions as higher-order values in `let` constructs and by `apply`, tuple and string literals, lambdas, built-ins (e.g., `minHood`, and `mux`), and the field calculus constructs `rep` and `nbr`.

Function definitions are prefixed by the `def` keyword, and they are visible by default only in the local module. In order for other modules to access them, the keyword `public` must be explicitly specified. Other modules can be imported, as well as Java static methods. Types are not specified explicitly: in fact, Protelis is duck-typed—namely, type-checked at run-time through reflection mechanisms. The language offers literals for commonly used numeric values, tuples, and strings. Instance methods can be invoked on any expression with the same "dot" syntax used in Java. Higher order support includes a compact syntax for lambda expressions, closures, function references, functions as parameters, and function application. Lastly, context properties, including device capabilities, are accessible through the `self` keyword. Environment variables can be accessed via the short syntax `env`.

Another relevant asset of Protelis is its recently developed library "protelis-lang" [91], streamlining the implementation of a number of algorithms found in the distributed systems literature. Among others, it includes several implementations of self-stabilising building block functions [92, 70], such as `distanceTo` to estimate distances, `broadcast` to send alerts, `summarize` to perform distributed sensing, and so on. Notably, the library also includes meta-machinery for "align-

---

[9]Alchemist is released as open source and available at `http://alchemistsimulator.github.io`

```
import protelis:coord:spreading // Import other modules
import java.lang.Math.sqrt // Import static Java methods
def privateFun(my, params) {
  my + params // Infix operators, duck typing
}
public def availableOutside() { // externally visible
  privateFun(1, 2); // Function call
  let aFun = privateFun; // Variable definition, function ref
  aFun.apply("a", "str"); // String literals, application
  let tup = [NaN, pi, e]; // Tuple literals, built-in numbers
  // lambda expressions, closures, method invocation:
  let inc3 = v -> {privateFun(v, tup.size())}
}
// MAIN SCRIPT
let myid = self.getDeviceUID(); // Access to device info
if (myid < 1000) { // Domain separation
  rep (x <- self.nextRandomDouble()) {// Stateful computation
    // Java static method call
    mux (sqrt(x) < 0.5) { // mux executes both branches
      // Library call, field gathering and reduction
      minHood(nbr(env.has("source")))
    } else { Infinity }
  } < 10
} else { // Mandatory else: every expression returns a value
  false // booleans
}
```

Figure 9: Example Protelis code showcasing a sampler of language features.

ing" aggregate computing programs along arbitrary keys, separating and mixing domains in a finer way than the `if` construct allows. These constructs, based on the `alignedMap` primitive of Protelis, enable highly dynamic meta-algorithms to be written, that open up new possibilities such as `multiInstance` [91], or allow for increased resilience and adaptation as in the case of `timeReplicated` [93].

Protelis is released as open source, and instructions on how to use it are available at http://www.protelis.org.

4.2. SCAFI: an API for the Scala programming ecosystem

From a pragmatic viewpoint, it is highly desirable to bridge the gap between field calculus-based DSLs and mainstream programming platforms and languages that embody, among other things, the functional, object-oriented, and actor-based paradigms (i.e., reference styles for in-the-small, in-the-large, and concurrent/distributed programming, respectively). Indeed, this may be critical to foster adoption, reducing accidental complexity through coherent syntax, semantics, and toolset, and paving the way to a more integrated programming experience.

External DSLs such as Protelis, despite the aid provided by DSL frameworks like Xtext [90], can require a lot of development and maintenance effort, since they must cover aspects ranging from language design to typing, and proper

```scala
trait FieldCalculus {
  def nbr[A](expr: => A): A
  def rep[A](init: => A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  def aggregate[A](f: => A): A

  // The following abstract access to the platform
  def mid(): ID
  def sense[A](name: String): A
  def nbrvar[A](name: String): A
}
```

Figure 10: SCAFI interface to the field calculus [94, 97].

tooling must be provided to enable full interoperability with the target platform in static, runtime, and debugging contexts. By contrast, internal DSLs are an interesting alternative, for they are expressed in the host language and are *de facto* equivalent to an API: they more seamlessly interoperate, and reuse the syntax, semantics, typing, and tools of their host language, at the expense of reduced flexibility due to the constraints exerted by the host environment.

Such considerations of pragmatism, reuse, and interoperability motivate SCAFI (Scala Fields) [94], an aggregate computing framework including a field-calculus DSL internal to the Scala programming language [95], also integrated into the Alchemist meta-simulator [96], as well as an actor-based platform for distributed aggregate systems [97, 98]. The choice of Scala as the host language was inspired by its *(i)* interoperability across the JVM platform, *(ii)* seamless integration of the object-oriented and functional paradigms, with support for lightweight component-based programming (cf., traits and self-types), *(iii)* advanced features for type-safe library development (cf., implicits, generic type constraints), *(iv)* syntax flexibility and sugar (cf., by-name arguments), allowing creation of fluent DSL-like APIs; and *(v)* prominent role in the scene of distributed computing frameworks (cf., Akka [99], Kafka [100], Spark [101]).

Complementarily, from the platform perspective the use of actor-based abstractions is instrumental to the integration of aggregate-level functionality into existing distributed systems (e.g., developed with more traditional techniques), by exposing collective coordination events and data through message or event-like interfaces [97].

In SCAFI, the field calculus is modelled through a Scala trait (i.e., an interface) like the one reported in Figure 10—where type parameters are specified in square brackets; syntax `=>T` denotes by-name parameters; syntax `T=>R` denotes function types; syntax `(.,.)` denotes tuples; and methods can be specified with multiple parameter lists. Interestingly, fields do not emerge at the type level. Indeed, with respect to HFC, SCAFI provides a slightly different semantics where neighbouring fields are substituted by a notion of "computation against a neighbour", which is carried out by "folding" over the set of aligned neighbours through a `foldhood` operation; coherently, `nbr` expressions can only be used

26

within the `expr` expression of the fold.

In practice, writing an aggregate program is as simple as subclassing `AggregateProgram` and defining a `main` method which represents the entry point of the round logic. Operationally, an `AggregateProgram` instance acts simply as a function from an abstract `Context` to an `Export`. Hence, for a platform to support local execution of field computations it is just a matter of instancing an aggregate program (possibly mixing in components to provide access to platform-level functionality), preparing contextual information (i.e., previous state, sensor data, and messages from neighbours), and running a computation round according to the device lifecycle.

Working with a general-purpose, multi-paradigm programming language like Scala can give developers quite a lot of flexibility and power with regards to the design and implementation of field libraries and programs. Indeed, generic, object-oriented, and modular programming techniques are used within SCAFI and its standard library to provide type-safe, expressive, reusable functionality.

Figure 11 shows an example of programming in SCAFI, including the definition of a reusable block `G` (extending distance calculation [1, 70]), the import of functionality through *mix-ins* (`with`), the use of type-class-style assumptions on arguments via context bound "`[V: Bounded]`" for implicit resolution [102], and pattern matching "`case...=>...`". Despite this power, some care has to be taken when mixing standard Scala features with SCAFI code: because field computations build on a notion of alignment [71] for correct composition, their operation can be disrupted by features that locally affect or alter the abstract position of field construct calls in the program, such as by-need constructs, control structures like Scala's `if`, iterative constructs, and operations on collections (especially when these are lazy).

In addition to the DSL, SCAFI also provides an actor-based platform [97], implemented on top of the Akka toolkit [99], to ease the development of distributed aggregate systems. It currently supports two architectural styles [98]: *(i) a fully peer-to-peer style*, where individual devices have everything they need to make up a system through decentralised interaction; and *(ii)* a client-server style, where device-to-device interaction leverages a central server as an intermediary.

SCAFI is released as open source, with instructions on how to obtain and use it available at `http://scafi.github.io`.

4.3. Aggregate Programming

Building upon these theoretical and pragmatic foundations, aggregate programming [1] elaborates a layered architecture that aims to dramatically simplify the design, creation, and maintenance of complex distributed systems. This approach is motivated by three key observations about engineering complex coordination patterns:

- composition of modules and subsystems must be simple and transparent;

- different subsystems need different coordination mechanisms for different regions and times;

27

```scala
trait BlockG { // Component
  self: FieldCalculus with StandardSensors => // Dependencies

  // Generic function with type-class constraint on V
  def G[V: Bounded](source: Boolean,
                    field: V,
                    acc: V => V,      // Function type
                    metric: => Double // By-name parameter
                   ): V =            // Return type
    rep((Double.MaxValue, field)) {
      case (dist, value) => // Function by pattern matching
        mux(source) {
          (0.0, field) // Tuple syntax sugar for Tuple2(_,_)
        }{
          minHoodPlus { // Requires (Double,V) to be Bounded
            (nbr { dist } + metric, acc(nbr { value }))
          }
        }
    }._2 // Selects 2nd element of tuple
}

class Program extends AggregateProgram
              with StandardSensors with BlockG { // Mixins
  def main: Double = // Program entry point
    distanceTo(isSource)

  def isSource = sense[Boolean]("source")

  def distanceTo(source: Boolean): Double =
    G(source, 0.0, _ + nbrRange, nbrRange)
}
```

Figure 11: Example of SCAFI DSL code.

- mechanisms for robust coordination should be hidden by abstractions, such that programmers are not required to interact with the details of their implementation.

Field calculus (along with its language incarnations) provides mechanisms for the first two, but is too general to guarantee resilience and too mathematical and succinct in its syntax for direct programming to be simple: some methodology is needed to properly scale with complexity.

Aggregate programming thus proposes two additional abstraction layers, as illustrated in Figure 12, for hiding the complexity of distributed coordination in complex networked environments. First, the "resilient coordination operators" layer plays a crucial role both in hiding the complexity and in supporting efficient engineering of distributed coordination systems. First proposed in [92], it is inspired by the approach of combinatory logic [103], the catalogue of self-organisation primitives in [104], and work on self-stabilising fragments of the field calculus [79, 70, 80]. Notably, three key operators within this self-stabilising fragment cover a broad range of distributed coordination patterns: operator G is
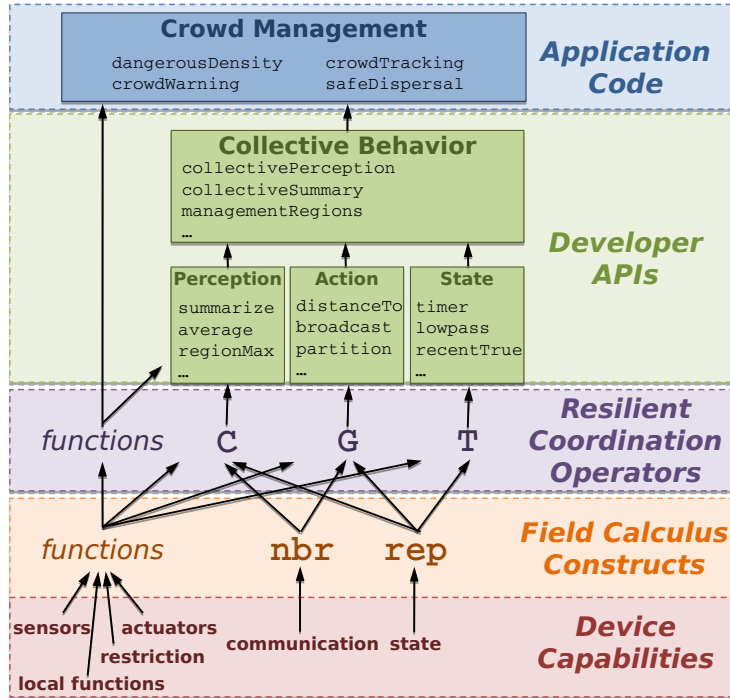
Figure 12: Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT (Internet-of-Things) systems. Figure adapted from [1].

a highly general information spreading and "outward computation" operation, `C` is its inverse, a general information collection operation, and `T` implements bounded state evolution and short-term memory.

Above the resilience layer, aggregate programming libraries [91, 81] capture common patterns of usage and more specialised and efficient variants of resilient operators to provide a more user-friendly interface for programming. This definition of well-organised layers of abstractions with predictable compositional semantics thus aims to foster *(i) reusability*, through generic components; *(ii) productivity*, through application-specific components; *(iii) declarativity*, through high-level functionality and patterns; *(iv) flexibility*, through low-level and fine-grained functions; and *(v) efficiency*, through multiple components with coherent substitution semantics [70, 81].

Within these two layers, development has progressed from an initial model built only around the spreading of information to a growing system of composable operators and variants. The first of these operator/variant families to be developed centred around the problems of spreading information, since in-

teraction in aggregate computing is often structured in terms of information flowing through collectives of devices. A major problem thus lies in regulating such spreading, in order to take into account context variation, and in rapidly adapting the spreading structure in reaction to changes in the environment and in the system topology. Here, the gradient (i.e., the field of minimum distances from source nodes) in its generalised form in the G operator is what captures, in a distributed way, a notion of "contextual distance" instrumental for calculating information diffusion, and forms the basis for key interaction patterns, such as outward/inward bounded broadcasts and dynamic group formation, as well as higher-level components built upon these.

The widespread adoption of gradient structures in algorithms stresses the importance of fast self-healing gradients [105], which are able to quickly recover good distance estimates after disruptive perturbations, and more "dependable" gradient algorithms in which stability is favoured by enacting a smoother self-healing behaviour [106]. Several other alternative gradient algorithms have also been developed, addressing two main issues. Firstly, the recovery speed after an input discontinuity, which has first been bounded to $O(diameter)$ time by the CRF (constraint and restoring force) gradient algorithm [105], further improved to optimal for algorithms with a single-path communication pattern by the BIS (bounded information speed) gradient algorithm [107, 108], and refined to optimality for algorithms with a multi-path communication pattern by the SVD (stale values detection) gradient algorithm [109]. Secondly, the smoothness and resilience to noise in inputs, first addressed by the FLEX (flexible) gradient algorithm [106] and then refined and combined with improved recovery speed by the ULT (ultimate) gradient algorithm [109].

To empower the aggregate programming tool-chain, other building blocks have been proposed and refined in addition to gradients: consensus algorithms [110], centrality measures [111], leader election and partitioning [92], and most notably, *collection* [70, 81]. The collection building block C progressively aggregates and summarises values spread throughout a network into a single value, e.g., their sum or other meaningful statistics. Based itself on distance estimation through gradients, a general *single-path* collection algorithm has been proposed in [92] granting self-stabilisation to a correct value, then *multi-path* collection has been developed for improved resiliency in sum estimations [70], and finally refined to *weighted multi-path* collection [112] and its parametric extension [113], which is able to maintain acceptable whole-network sums and maxima even in highly volatile environments. A different approach to collection has also proved to be effective for *minimum/maximum* estimates: overlapping replicas of non-self-stabilising *gossip* algorithms [93] (with an appropriately tuned interval of replication), thus combining the resiliency of these algorithms with self-stabilisation requirements.

In sum, the current state of aggregate computing features pragmatic implementations of field calculus well-integrated into modern languages. These in turn support an expanding library of resilient building blocks with various tradeoffs in their dynamical behaviour, and which can be used as the basis for implementation of a wide variety of distributed applications.

5. Perspectives and Roadmap

Over the past decade, aggregate computing has moved from a fragmented collection of ideas and tools to a stable core calculus and a coherent layered framework for the engineering of distributed systems. Thus, even as the underlying theory continues to be developed, as shown in [114], a significant portion of research and development can shift to more pragmatic issues linked to applications and higher levels of the aggregate computing stack. In this section, we review a number of such research directions, which include elaboration of libraries (Section 5.1), techniques to control dynamics (Section 5.2), management of mobile devices and processes (Section 5.3), development of software platforms (Section 5.4), addressing non-functional requirements such as safety and security (Section 5.5), and applications (Section 5.6).

5.1. Elaboration of Libraries

The most immediate and incremental line of future development for aggregate computing is the elaboration of the existing collection of libraries, to form a more broadly applicable and easier to use interface at the top of the aggregate computing stack. One of the key directions for such additions and refinements will be the development of alternative implementations of core resilient building block algorithms. The current resilient building block algorithms were selected to be both simple, in order to make composition proofs more tractable, and highly general, which comes at a cost of being unable to make assumptions about application needs or network conditions. As such, these algorithms are also often much lower performance than they might be, and in most circumstances a software engineer would prefer to be able to use more sophisticated and/or more specialised alternatives. Development of such alternatives has already begun as described above (e.g., [113, 108, 93]), but there is much opportunity for further development and for adaptation of existing high-performance algorithms into the aggregate programming framework.

Complementarily, despite the breadth of the core building blocks, there are also many distributed algorithms whose behaviour cannot be reasonably expressed in terms of these building blocks. Another direction of future library elaboration will thus be the incorporation of a larger range of widely used distributed algorithms (e.g., those in [115] and [76]). Similarly, particular application domains will suggest adaptation or development of more specialised collections that capture the common design patterns and necessary functionalities peculiar to a domain. Adapting pre-existing algorithms into an aggregate programming context will often pose some challenges, however, as most prior distributed algorithms are not self-stabilizing and/or have not been designed with composition in mind.

Overall, the process of library elaboration is expected to follow a natural incremental progression of ongoing maturation and professionalisation, driven by issues discovered as the other lines of future development outlined below exercise the existing libraries to expose their current shortcomings and needs for enhancement.

31

5.2. Understanding and Controlling Dynamics and Feedback

Much of the work to date on aggregate computing has focused on the converged properties of a system, such as self-stabilisation [76, 70] and eventual consistency [82]. These theoretical approaches, however, assume that the network of devices is often in a persistent quasi-stable state in which the set of devices, their connections to one another, and their environment all do not change for a significant length of time. In large scale systems, however, such quasi-stable states are typically rare and short-lived: there is almost always something changing with respect to some device, thus constantly injecting perturbations into the system. Prior compositional safety analysis regarding self-stabilisation and eventual consistency also does not apply in the case of systems involving feedback, and many applications do require feedback either directly between building blocks or indirectly via interactions with the environment.

The control theory literature has many well-developed tools for analysing the response of complex systems under perturbation and in the presence of feedback, including Lyapunov stability theory [116], passivity theory [117, 118, 119], center manifold theory [120, 121], the Perron-Frobenius Theorem [122], and small-gain stability [123, 124, 125]. The mathematical frameworks for such tools are not straightforward to adapt for application to aggregate computing building blocks. With careful work, however, they may often still be applied, e.g., through identification of appropriate Lyapunov functions to bound the convergence behaviour of a building block.

Early work in this area shows promise, enabling analysis and prediction of aggregate computing systems with feedback between building blocks [126, 127] and providing stability analysis and tight convergence bounds for particular applications of the G operator [128, 129] and C operator [130]. An important area for future development is thus to expand these results to cover a large sublanguage of aggregate computing systems and to apply them in order to refine and improve the dynamical performance of building blocks.

A potential complementary approach to these problems is to instead apply runtime verification techniques in order to control the behaviour of an aggregate system, possibly exerting some kind of corrective action or feedback to maintain a given quality of service. Early work in this area [131] shows that the field calculus is a promising language for expressing runtime properties to be monitored (which may in turn be expressed either in field calculus or in other formalisms). The recent development of the share construct for optimal state diffusion [132] further supports this claim, by providing means to check temporal properties without delays. Notably, runtime verification methods are often too expensive to be used on the complex state spaces of distributed systems, leading to the development of state-reduction methods like mean-field approximation [133, 134]; field calculus may provide an alternative method for state reduction that is more readily able to be applied to a broader class of systems. Future developments may provide automatic translations of properties expressed in spatio-temporal logics into field calculus, possibly improving over existing similar approaches in the field of runtime verification of distributed systems.

5.3. Mobility of Devices and Processes

Another key area for expansion of aggregate computing, both in theory and practice, is better handling of mobility, both of devices and of processes dispersed through networks of devices. From a theoretical perspective, this is closely interwoven with the need for a deeper understanding of convergence dynamics, as systems with mobile devices or processes typically do not ever achieve the quasi-stable states required for self-stabilisation to hold. Work to date, however, has instead typically depended on the informal observation that "slow enough" mobility does not disrupt commonly used self-stabilising building blocks—an assumption called into question by the results in [130]. Theoretical work is needed to predict and bound regions of stability and effects of perturbation, as well as to develop improved building block alternatives for conditions where the identified dynamics are unsatisfactory. There is also a need to expand the existing building block libraries to support applications involving mobility. For controlling the physical motion of devices, a number of building blocks have been demonstrated or proposed throughout the swarm robotics and multi-agent systems literature, including a number already formulated as building blocks for aggregate computing (e.g., [135, 136, 137]).

Complementarily, another direction deserving exploration concerns the ability of the field calculus to effectively express a dynamical collection of concurrent field computations with possibly dynamic domains. In this case, it is not the device that is the focus of mobility, but instead code and processes that dynamically deploy, migrate, upgrade, and terminate during system operation, as considered for example in [1, 138, 93] [139]. To effectively support mobility in aggregate computing, the large volume of prior work on algorithms and strategies for such systems needs to be systematised and organised, analysed for compositional safety and bounds on convergence, and adapted for use in aggregate computing based on the results of analysis.

While some initial work has been reported in [91], there is need for development of a reasoned stack, from fine-grained alignment primitives to meta-algorithms, that neatly increases the practical expressiveness of the field calculus and better captures the dynamism, transitoriness and opportunistic traits of forthcoming IoT scenarios [140] [141]. Moreover, novel support for the meta-management of field computation domains could help in defining dynamic *coalitions* or *teams* [51] of devices—i.e., short-lived, goal-directed groupings attempting to maximise individual or group utility, respectively.

5.4. Software Platforms

Aggregate computing targets a broad range of application scenarios, generally characterised by inherent distribution, heterogeneity, mobility, and lack of stable infrastructure (including computation, storage, and networking media). Development, deployment, and runtime management of such applications can be greatly facilitated by development of middleware or similar software platforms [98]. Middleware is a long-established approach to injection of abstraction layers between application software and underlying software, hardware, or network challenges [142], providing a means of sharing and layering functionality to

assist with distributed systems challenges such as security and authentication, privacy and information management, run-time monitoring, fault tolerance, etc. Middleware does not solve or isolate these problems, particularly with where it regards security and safety, but provides a means of at least sharing and reusing patterns and means of addressing them.

Though middleware is used throughout the world of distributed computing, there are some issues (e.g., those discussed in this section, like mobility and control) and opportunities specifically related to aggregate computing and coordination that deserve attention. In particular, note that the aggregate programming model is partially declarative in that it abstracts over a number of details such as, for instance, the specifics of neighbourhood-based communication and the order and frequency of micro-level activities sustaining application execution—details that can be delegated to corresponding platform services for topology management, scheduling, and round execution. This abstraction provides a high degree of flexibility for the actual platform implementation, which is free to apply optimisations of various sorts, from simpler (e.g., avoiding broadcasting redundant messages) to more complex ones (e.g., mixing of different communication modes).

Indeed, the entire aggregate computing system can be run according to varying strategies, depending on the pragmatics of communication and underlying hardware [98]. At one extreme, programs may be executed in a fully distributed peer-to-peer environment, where end-devices directly communicate to peer neighbours and each runs its own fragment of aggregate logic. At the other extreme are completely centralised solutions where end-devices act only as managers for sensors and actuators, sending perceptions upstream to one or more servers that run computations on their behalf and ultimately propagate actuation data downstream.

Crucially, this flexibility suggests that aggregate computing may enable development of a more principled spectrum for transitioning between cloud systems and distributed systems, embracing as well the emerging domains of edge and fog computing [143, 144, 145] [146]. Aggregate computing may thus enable adaptive adjustment of systems for opportunistic and QoS-driven exploitation of available infrastructural resources, as well as to the intrinsic adaptation required to deal with emerging IoT scenarios. For instance, an aggregate system specification can be mapped to a system of actors [97] where each actor is responsible for a specific aspect of the overall computation and communication and can be migrated to different machines while preserving coordination by automatically adapting the bindings [98]. Much work remains, however, to further develop both the theory of adaptive execution and to put such execution into practice.

5.5. Non-functional requirements

In real-world engineering efforts, the effort required to make a system that addresses core functional requirements is often heavily outweighed by the effort required to deal with additional considerations such as safety, security, privacy,

and sustainability. The success or failure of aggregate programming as an approach to distributed systems engineering is thus likely to depend strongly on whether its implementation is sufficiently able to either help address such non-functional requirements or at least not interfere with other efforts to address them.

5.5.1. Safety

Safety in software usually concerns protections included in software to prevent, intercept, and react to *unintentional* harm. In the context of aggregate programming, safety is relevant at several levels: platform-level safety, language-level safety, algorithmic safety, and compositional safety.

Platform safety is not a property of aggregate programming itself, but rather is inherited from specific implementation and deployment. As discussed in Section 5.4, in fact, aggregate programming abstracts away from the middleware in charge of allowing network communications. Such middleware, however, will in the end be part of the deployed system, and its safety properties (including possible issues) will propagate to the aggregate system.

Language safety refers on the safety of the specific implementation of field calculus. One key element of field calculus is alignment [38, 71], which determines whether or not two devices belong to the same domain at some specific point during the execution. Practical implementations must deal with alignment [71]. The issue is usually tackled by annotating shared values in such a way that the same computation path allows reconstruction of the annotation. However, both the existing implementations have potential issues due to their hosting platform: in Protelis, alignment is not applied when Java code is called from within the DSL, as Java libraries are not aware of the requirement to build annotations; in SCAFI, the primitive construct `aggregate` must be used to wrap lambdas to turn them into "aggregate functions", i.e., function objects that are "tagged" with unique identifiers when they are created and hence respect alignment when invoked (by adding a corresponding labelled node to the evaluation tree). These aspects call for additional care when designing interaction between aggregate programming languages and their host platform, and might be further improved in future implementations. Language safety also includes language features that help developers write correct programs, such as type checking, debugging tools, null safety, and so on.

Algorithmic safety is related to the guarantees that algorithms offer, especially those more frequently used and those included in standard libraries. Eventual consistency [82], self-stabilisation [70], and self-adaptation to device distribution [147] are good examples of algorithmic safety guarantees. Further relevant algorithmic safety information includes behaviour with respect to time, which is particularly relevant for hard real-time systems [83]. One promising research line to obtain further safety information of existing and novel aggregate algorithms is analysis conducted through the tools classically leveraged by control theory [126, 128, 129, 130], as described in Section 5.2.

Compositional safety, finally, refers to the conservation of safety properties when algorithms are combined. Existing work has identified a collection

35

of fundamental building blocks that propagate their algorithmic safety properties when combined in non-cyclic constructions [92]. Current work, however, provides little in the way of compositional safety guarantees regarding time or systems involving feedback mechanisms.

5.5.2. Security

Security refers to the ability of the system to prevent, detect, monitor, and react to *intentional* malicious attacks. Security is a critical concern in computer science in general and especially in open environments, such as those envisioned in pervasive computing and IoT scenarios involving vast numbers of devices administered by individuals and organisations with no particular knowledge of security. This problem is multifaceted and requires carefully thought, full-stack solutions that also consider orthogonal requirements, such as the cost of security-related computational tasks in resource-constrained devices.

Similarly to safety issues, security issues arise at every level of the computing platform: a hypothetical perfectly secure aggregate language used to write only demonstrably secure programs can still be exploited by attackers if the platform hosting the computation (virtual or hardware) or enabling communication is not secure. At the same time, a perfectly secure middleware does not guarantee security at the higher abstraction levels, since the mechanics of the aggregate system or implementation shortcomings could be leveraged to induce unwanted behaviours.

Regarding application-level interaction, since coordination activity in aggregate computing is substantially based on a premise of cooperation between the participating entities, it is often sensitive to attacks that may trigger epidemic deviation. That is, what is the extent to which agents and their data can be trusted? In order to assess and mitigate the impact of voluntary or involuntary misbehaviour, adoption of computational trust has proven useful [148] and applicable even in decentralised settings, in which no central authority is available to certify recipients and endpoints, and in scenarios where seamless opportunistic interaction is the norm. The proposed system provides some degree of protection from malicious attackers with minimal requirements (i.e., without any infrastructural service or assumption about other devices), but also requires special, trust-aware versions of algorithms, as well as a non-trivial parameter tuning process. Combined, these restrictions limit the applicability of the proposed techniques.

Aggregate computing security at the language and language implementation level is, to the best of our knowledge, currently unexplored. One possible approach to tackling the issue is by performing design and code level hardening [149], which would require a specific analysis phase oriented to threat identification and risk calculation, before the design of potential countermeasures.

A number of security issues, not strictly related to coordination but of prominent importance in real-world, trustworthy systems, can be addressed in the middleware layer and through proper deployment solutions. For example, support is needed to enable safe code mobility and execution, as proposed in [85],

which may be required in scenarios characterised by significant dynamicity requirements or demands for automatic deployment of new functionality. Additionally, despite the decentralised and inherently scalable nature of aggregate systems, availability issues need to be considered, according to the specifics of applications, especially with respect to nodes playing a crucial role in algorithms (e.g., sources, hubs, collectors, region leaders). Security at the platform level is usually delegated almost entirely to the platform itself. However, some specific security features of specific platforms may have relevant consequences for aggregate software executing on them. For instance, in [150] a number of attacks are designed to compromise an aggregate program executing a trust-aware [148] version of an aggregate algorithm. Interestingly, a class of attacks known as Byzantine behaviours [151], which include *selective attacks* (sending data only to selected targets, or sending different information to different neighbours), *masquerade attacks* (imitating the identity of another device), and *Sybil attacks* (faking multiple identities) could actually exploit the trust-based protection mechanism via carefully crafted messages in order to impair the aggregate system. A possible solution is leveraging the BlockChain technology [152] at the middleware level to provide transparent protection (i.e., without impact on application logic) from Byzantine attacks. Unfortunately, permissioned BlockChains require certification authorities to provide identities and roles, hence reducing system openness, while permissionless BlockChains pose serious limitations on throughput (of transactions corresponding to coordination messages, and hence to system reactivity).

As practical aggregate programming platforms come to target embedded devices (as they are likely to do in future), more focus will be required on lower level platform issues, especially if the protection layer usually offered by the operating system will not be available or will be subject to more severe limitations due to efficiency requirements. In this context, even "close-to-metal" attacks such as Meltdown [153], Spectre [154], and (Th)Rowhammer [155, 156] will need to be taken into account when selecting or designing the execution platform.

5.5.3. Privacy

Another related theme is privacy and confidentiality of information. The privacy properties of the data propagated and collected by aggregate programs needs to be understood and guaranteed, or else participation may be hindered. Privacy concerns have not, so far, been strongly considered in designing and implementing aggregate programming languages or programs. In many ways, these concerns overlap with security: for example, the attacks mentioned in Section 5.5.2 have the potential not just to disrupt computations, but to be used to maliciously extract data not meant to be available. Another issue that must be tackled in the future includes adequate encryption not just of data exchanged between communicating devices (an issue that could arguably be tackled at the platform level), but of unauthorised access to some portions of this data. Due to alignment, there may be some portions of a collection of data that are sent to all neighbours, but are meant to be accessed only by a subset of them, due to

domain separation induced by a distributed branching construct. Currently, in existing practical aggregate languages, nothing prevents legitimate participants to the system, even running on some theoretical perfectly secure platform, from accessing all of the data shared by any of the neighbours. Adequate protection of confidentiality at this level will require novel research and careful thought.

5.5.4. Efficiency and sustainability

Efficiency of computing is already a major concern for platforms with constrained energy budgets (e.g., those powered by batteries and/or energy harvesting), and sustainability is becoming of increasing concern as a non-functional requirement for computing systems [157]. Sustainability and efficiency, which are closely related, are pervasive issues requiring special attention, as they can pose limitations to the techniques that can be deployed in order to satisfy both functional and other non-functional requirements. For example, encryption techniques intended for tackling privacy and security issues may require the use of computational resources that may imply unacceptably high battery consumption. Moreover, network and power efficiency concerns are often among the elements that are simplified away in simulated models, shifting the burden down the line to deployment and implementation. As discussed for safety in Section 5.5.1, efficiency should also be considered as a cross-cutting concern throughout the whole stack of aggregate computing. Besides algorithmic efficiency, which has a great deal of impact, there are also a number of concerns related to language and platform efficiency.

Although some of the antecedents of aggregate computing have been quite energy efficient [158], the increased generality and functionality of recent platform implementations has as a byproduct resulted in less efficient implementations. Aggregate programming has been recently experimented with in conjunction with long-range, low-power communication devices (LoRaWAN [159]), yielding mixed results and suggesting future research directions [160]. One relevant issue for integration in such networks is the fact that both existing practical implementations of the field calculus require a Java Virtual Machine to execute, which is far above the computational resources of small microcontrollers such as those commonly found in LoRa nodes. Consequently, such nodes were used as long range networking interfaces for more computationally capable devices; such a configuration, however, would make sense only for a narrow range of applications, where either battery power is not a relevant concern, or the computation and communication system power consumption is only a small fragment of the overall consumption (e.g., with self-propelled mobile devices), and would likely complicate the deployment of self-powered LoRa systems [161]. Future efforts devoted to providing highly efficient implementations of the field calculus targeting microcontrollers, however, could possibly once again make aggregate programming of ensembles of low power devices feasible. A second relevant factor limiting the adoption of aggregate programming over long range networks is its use of network capacity: unlike the antecedent implementations in Proto [18], current field calculus implementations do not have bandwidth capacity saving systems in place (such as a mechanism not to send identical

messages in sequence), nor do they focus on reducing message size. At present, the produced packet size produced is large enough to prevent any non-trivial aggregate programming application from using long-range, low-power communication [160], particularly if packets contain Java objects transmitted with default serialization. Unfortunately, optimisations focused on reducing message size often conflict with optimisations oriented at improving simulation performance and accessibility of debugging information, as well as frequently conflicting with privacy and security requirements. The issue might be mitigated in a number of ways, however, including isolating the construction of network packets and of the annotations used to fetch data based on the current computation, and by providing multiple implementations that target different efficiency tradeoffs.

5.6. Applications and Pragmatics

Finally, the core goal for the aggregate computing research thrust has always been to enable simpler, faster development of more resilient distributed applications. Having developed both its theoretical foundations and the layered system of algorithms and libraries exploiting those foundations, one of the major directions of current and future work is indeed to apply these developments to real-world problems across a variety of domains.

5.6.1. Pervasive computing, IoT, smart-cities

One key application area, previously discussed in [1] and other works, is pervasive or IoT scenarios in dense urban environments. As the density of communicating devices increases, their interactions put pressure on the available fixed infrastructure and the opportunities for local interaction increase [162]. This is particularly acute during transient events when demand and the available infrastructure become mismatched, such as during festivals or sporting events when the number of people packed into an area spikes, or during natural disasters and other emergencies when the available infrastructure may be degraded. One of the critical challenges for such applications is simply to access the potential peer-to-peer capabilities of devices, which are often closed platforms and are currently typically configured primarily for asymmetrical communication with fixed infrastructure or individually connected personal networks. These constraints are both loosening over time as app infrastructures continue to spread and develop on many platforms. Finally, the benefits of distribution must be effectively balanced with tight energy budgets on many devices and the continuous value of non-local interactions enabled by cloud connections.

A closely related research problem revolves around the coordination of computation activities across the edge, fog, and cloud layers. Edge and fog computing are emerging paradigms that complement traditional cloud services—provided by massive, remote data centres—with elastic resource provisioning "at the edge of the network" [163, 164], close to where computation inputs are taken and computation outputs are needed (i.e., to users). The problem forces that have motivated this evolution are essentially the same as those that originated the *spatial computing* movement [57]. Namely, at their basis is the

realisation of the key role of physical locality, as location in space affects both the performance and the feasibility of computation—cf., latency-sensitive applications, connectivity limitations. The combined action of miniaturisation, dense deployments, and heterogeneity promotes a vision where more and more resource-constrained devices offload computations locally, rather than through global connectivity. Arguably, by its intrinsically spatial nature [165] and its declarativity [98], aggregate computing may help to define edge/cloud computing ecosystems where both locality and increasingly non-local aspects are taken into account [146, 141].

5.6.2. Robot swarms, unmanned aerial vehicles

Another important emerging application area is control of drones and other unmanned vehicles, driven by the rapidly increasing availability of high-quality platforms at various levels of cost and capability. With the emergence of highly capable autopilots, the need for detailed human control is decreased and it becomes desirable to shift from the current typical practice of multiple people commanding a single platform toward a single person controlling many platforms. Aggregate computing is a natural fit for approaching multi-platform control, using paradigms such as those discussed in [135] and [136]. In implementation, however, the challenges of mobility become acute as one considers rapid physical movements. Likewise, a better understanding of convergence dynamics and feedback will be needed. Work in this space will also demand significant elaborations in aggregate computing libraries, adapting manoeuvres from the applicable literature and doctrine into additional composable building block components. Finally, there are also major pragmatic issues to be addressed in platform interfaces, including a plethora of standards, safety issues, and appropriate incorporation of resource and manoeuvring constraints.

5.6.3. Agent-based planning

Agent-based planning uses similar principles, computing plans for future actions over an aggregate of agents. This generalisation, however, typically also connects representations of future plans, tasks, goals, and environment into the aggregate [138], as some combination of additional virtual devices in the aggregate and virtual fields that devices can interact with. Examples include the poly-agent approach to modelling and planning [166] and agent-based sharing of airborne sensors [167, 168]. When agent-based planning is centralised, managing projections and tasks is straightforward; when distributed across physical agents, however, there are important questions to be addressed regarding where projections and tasks should be hosted, to what degree they should be duplicated, and how to synchronise information between duplicates.

5.6.4. Networked systems management

Aggregate computing can also be applied to more conventional networked systems. In this case, the links between neighbours are defined by (not particularly spatial) physical network connections, virtual network relationships such

as in an overlay network, or else logical relationships such as interaction patterns between services. As long as the number of such neighbours is relatively constrained, such that sending regular updates to neighbours is not problematic, many of the same sorts of coordination approaches that work in other application areas can work in areas such as these as well. Examples of applications in this space include coordinating recovery operations for networks of enterprise services [89], coordinating a checkpoint-based "rewind and replay" across interacting services to undo the effects of a cyber-attack [169], and integrating applications across intermittently connected distributed cloud nodes [169]. In this domain, in most cases it is not cost-effective to try to write or refactor entire services and applications into an aggregate computing paradigm. Instead, aggregate computing appears better used as a meta-level coordination and control service, helping to determine things like when and where to migrate services across machines, how many instances of a service should be used, how to rendezvous between services that need to communicate, and so on. Future work in this space is thus likely to focus on extending libraries to better support various coordination paradigms, particularly with distributed graph algorithms for supporting coordination regarding dependencies and information flows, and on the pragmatics of interfacing with complex legacy applications.

5.6.5. Other application domains

In addition to the domains presented here, aggregate computing offers potential value in many other application domains as well: it is likely to offer value in any domain with an increasing number and potential volatility in collections of devices capable of communicating locally. The ongoing continuation of miniaturisation and embedding of computational devices means this is likely to apply in most areas of human endeavour, to one degree or another. Across all such domains, just as in the four domains described in detail, it is likely to be the case that aggregate computing will not be the focus of the system but rather, much like any other specialised library, used as a modular component: and most specifically, as a component providing a *coordination service*. A critical challenge for the future, then, will be to continue shaping and improving libraries and interface patterns in response to the needs of these application domains, in order to allow aggregate computing to become as invisible as possible in the actual process of systems engineering.

6. Conclusions

Aggregate computing is a potentially powerful approach to the engineering of distributed systems, emerging from the distillation of a wide variety of approaches to coordination into the field calculus. This mathematical core then serves as the basis for a layered approach to pragmatic development of composable and resilient distributed systems. The future of aggregate programming involves both continued development of its core theoretical tools as well as work to realise its potential across a wide range of important application domains.

Bibliography

[1] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the Internet of Things, IEEE Computer 48 (9) (2015) 22–30 (2015). `doi:10.1109/MC.2015.261`.

[2] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: International Conference on Coordination Models and Languages (COORDINATION), Vol. 10852 of Lecture Notes in Computer Science, Springer, 2018, pp. 252–279 (2018). `doi:10.1007/978-3-319-92408-3_12`.

[3] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems (TOPLAS) 7 (1) (1985) 80–112 (1985). `doi:10.1145/2363.2433`.

[4] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices, The Jini Technology Series, Addison-Wesley Longman, 1999 (1999).

[5] P. Wyckoff, S. W. McLaughry, T. J. Lehman, D. A. Ford, T Spaces, IBM Journal of Research and Development 37 (3 – Java Techonology) (1998) 454–474 (1998). `doi:10.1147/sj.373.0454`.

[6] A. Brogi, P. Ciancarini, The concurrent language, Shared Prolog, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (1) (1991) 99–123 (1991). `doi:10.1145/114005.102807`.

[7] N. H. Minsky, V. Ungureanu, Law-Governed interaction: A coordination and control mechanism for heterogeneous distributed systems, ACM Transactions on Software Engineering and Methodology (TOSEM) 9 (3) (2000) 273–305 (2000). `doi:10.1145/352591.352592`.

[8] G. Cabri, L. Leonardi, F. Zambonelli, MARS: A programmable coordination architecture for mobile agents, IEEE Internet Computing 4 (4) (2000) 26–35 (2000). `doi:10.1109/4236.865084`.

[9] A. Omicini, E. Denti, From tuple spaces to tuple centres, Science of Computer Programming 41 (3) (2001) 277–294 (2001). `doi:10.1016/S0167-6423(01)00011-9`.

[10] R. Menezes, J. Snyder, Coordination of distributed components using LogOp, in: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Vol. 1, CSREA Press, 2003, pp. 109–114 (2003).

[11] I. Merrick, A. Wood, Scoped coordination in open distributed systems, in: International Conference on Coordination Languages and Models, Springer, 2000, pp. 311–316 (2000). `doi:10.1007/3-540-45263-X_21`.

[12] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: A kernel language for agent interaction and mobility, IEEE Transaction on Software Engineering (TOSE) 24 (5) (1998) 315–330 (1998). `doi:10.1109/32.685256`.

[13] A. L. Murphy, G. P. Picco, G.-C. Roman, LIME: A coordination model and middleware supporting mobility of hosts and agents, ACM Transactions on Software Engineering and Methodology (TOSEM) 15 (3) (2006) 279–328 (2006). `doi:10.1145/1151695.1151698`.

[14] M. Viroli, D. Pianini, J. Beal, Linda in space-time: An adaptive coordination model for mobile ad-hoc environments, in: 14th International Conference on Coordination Models and Languages (COORDINATION), 2012, pp. 212–229 (2012). `doi:10.1007/978-3-642-30829-1_15`.

[15] R. Tolksdorf, R. Menezes, Using swarm intelligence in Linda systems, in: Engineering Societies in the Agents World IV, Vol. 3071 of Lecture Notes in Computer Science, Springer, 2004, pp. 519–519 (2004). `doi: 10.1007/978-3-540-25946-6_3`.

[16] M. Viroli, M. Casadei, S. Montagna, F. Zambonelli, Spatial coordination of pervasive services through chemical-inspired tuple spaces, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 6 (2) (2011) 14:1 – 14:24 (2011). `doi:10.1145/1968513.1968517`.

[17] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The TOTA approach, ACM Transactions on Software Engineering Methodologies (TOSEM) 18 (4) (2009) 1–56 (2009). `doi:10.1145/1538942.1538945`.

[18] J. Beal, J. Bachrach, Infrastructure for engineered emergence in sensor/actuator networks, IEEE Intelligent Systems 21 (2006) 10–19 (2006). `doi:10.1109/MIS.2006.29`.

[19] J.-L. Giavitto, C. Godin, O. Michel, P. Prusinkiewicz, Computational models for integrative and developmental biology, Tech. Rep. 72-2002, Univerite d'Evry, LaMI (2002).

[20] J.-L. Giavitto, O. Michel, J. Cohen, A. Spicher, Computations in space and space in computations, in: Unconventional Programming Paradigms, Vol. 3566 of Lecture Notes in Computer Science, Springer, Berlin, 2005, pp. 137–152 (2005). `doi:10.1007/11527800_11`.

[21] D. Coore, Botanical computing: A developmental approach to generating inter connect topologies on an amorphous computer, Ph.D. thesis, MIT, Cambridge, MA, USA (1999).

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

[23] K. Whitehouse, C. Sharp, E. Brewer, D. Culler, Hood: a neighborhood abstraction for sensor networks, in: 2nd International Conference on Mobile systems, applications, and services, ACM, 2004 (2004). `doi:10.1145/990064.990079`.

[24] M. Welsh, G. Mainland, Programming sensor networks using abstract regions., in: NSDI, Vol. 4, 2004, pp. 3–3 (2004).
URL `https://dl.acm.org/citation.cfm?id=1251178`

[25] W. Butera, Programming a paintable computer, Ph.D. thesis, MIT, Cambridge, USA (2002).

[26] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, P. Pillai, Meld: A declarative approach to programming ensembles, in: International Conference on Intelligent Robots and Systems (IROS), IEEE, 2007, pp. 2794–2800 (2007). `doi:10.1109/IROS.2007.4399480`.

[27] R. De Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: The SCEL language, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 9 (2) (2014) 7:1–7:29 (2014). `doi:10.1145/2619998`.

[28] M. Loreti, J. Hillston, Modelling and analysis of collective adaptive systems with CARMA and its tools, in: M. Bernardo, R. De Nicola, J. Hillston (Eds.), Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - 16th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2016, Bertinoro, Italy, June 20-24, 2016, Advanced Lectures, Vol. 9700 of Lecture Notes in Computer Science, Springer, 2016, pp. 83–119 (2016). `doi:10.1007/978-3-319-34096-8\_4`.
URL `https://doi.org/10.1007/978-3-319-34096-8_4`

[29] R. Hennicker, A. Klarl, Foundations for ensemble modeling—the Helena approach, in: Specification, Algebra, and Software, Springer, 2014, pp. 359–381 (2014). `doi:10.1007/978-3-642-54624-2_18`.

[30] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, Deeco: an ensemble-based component system, in: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering, ACM, 2013, pp. 81–90 (2013). `doi:10.1145/2465449.2465462`.

[31] F. Baude, L. Henrio, C. Ruz, Programming distributed and adaptable autonomous components—the GCM/ProActive framework, Software: Practice and Experience 45 (9) (2015) 1189–1227 (2015). `doi:10.1002/spe.2270`.

[32] M. Esteva, J.-A. Rodriguez-Aguilar, C. Sierra, P. Garcia, J. L. Arcos, On the formal specification of electronic institutions, in: Agent mediated electronic commerce, Springer, 2001, pp. 126–147 (2001).

[33] J. F. Hübner, J. S. Sichman, O. Boissier, Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels, International Journal of Agent-Oriented Software Engineering 1 (3/4) (2007) 370–395 (2007). `doi:10.1504/IJAOSE.2007.016266`.

[34] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, TAG: A Tiny AGgregation service for ad-hoc sensor networks, SIGOPS Operating System Review 36 (SI) (2002) 131–146 (2002). `doi:10.1145/844128.844142`.

[35] Y. Yao, J. Gehrke, The Cougar approach to in-network query processing in sensor networks, ACM Sigmod record 31 (3) (2002) 9–18 (2002). `doi:10.1145/601858.601861`.

[36] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, G. P. Picco, Mobile data collection in sensor networks: The TinyLime middleware, Elsevier Pervasive and Mobile Computing Journal 4 (2005) 446–469 (2005). `doi:10.1016/j.pmcj.2005.08.003`.

[37] R. Newton, M. Welsh, Region streams: Functional macroprogramming for sensor networks, in: Workshop on Data Management for Sensor Networks, 2004, pp. 78–87 (2004). `doi:10.1145/1052199.1052213`.

[38] G. Audrito, M. Viroli, F. Damiani, D. Pianini, J. Beal, A higher-order calculus of computational fields, ACM Transactions on Computational Logic 20 (1) (2019) 1–55 (jan 2019). `doi:10.1145/3285956`.

[39] D. Corkill, Blackboard systems, Journal of AI Expert 9 (6) (1991) 40–47 (1991).

[40] N. Busi, P. Ciancarini, R. Gorrieri, G. Zavattaro, Coordination models: A guided tour, in: Coordination of Internet Agents: Models, Technologies, and Applications, Springer, 2001, Ch. 1, pp. 6–24 (2001). `doi:10.1007/978-3-662-04401-8_1`.

[41] M. Viroli, A. Omicini, A. Ricci, Engineering MAS environment with artifacts, in: D. Weyns, H. V. D. Parunak, F. Michel (Eds.), 2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005), AAMAS 2005, Utrecht, The Netherlands, 2005 (26 Jul. 2005).

[42] A. Omicini, A. Ricci, M. Viroli, C. Castelfranchi, L. Tummolini, Coordination artifacts: Environment-based coordination for intelligent agents, in: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IEEE Computer Society, 2004, pp. 286–293 (2004).
URL `https://ieeexplore.ieee.org/document/1373490`

[43] M. Viroli, M. Casadei, A. Omicini, A framework for modelling and implementing self-organising coordination, in: ACM Symposium on Applied Computing (SAC), 2009, pp. 1353–1360 (2009). `doi:10.1145/1529282.1529585`.

[44] M. Casadei, M. Viroli, L. Gardelli, On the collective sort problem for distributed tuple spaces, Science of Computer Programming 74 (9) (2009) 702–722 (2009). `doi:10.1016/j.scico.2008.09.018`.

[45] D. Pianini, S. Virruso, R. Menezes, A. Omicini, M. Viroli, Self organization in coordination systems using a WordNet-based ontology, in: 4th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2010 (2010). `doi:10.1109/saso.2010.35`.

[46] M. Viroli, M. Casadei, Biochemical tuple spaces for self-organising coordination, in: Lecture Notes in Computer Science, Springer, 2009, pp. 143–162 (2009). `doi:10.1007/978-3-642-02053-7_8`.

[47] M. Viroli, On competitive self-composition in pervasive services, Science of Computer Programming 78 (5) (2013) 556 – 568 (2013). `doi:10.1016/j.scico.2012.10.002`.

[48] J. Ferber, G. Weiss, Multi-agent systems: an introduction to distributed artificial intelligence, Vol. 1, Addison-Wesley Reading, 1999 (1999).

[49] H. S. Nwana, L. C. Lee, N. R. Jennings, Coordination in software agent systems, British Telecom Technical Journal 14 (4) (1996) 79–88 (1996).

[50] Y. Cao, W. Yu, W. Ren, G. Chen, An overview of recent progress in the study of distributed multi-agent coordination, IEEE Transactions on Industrial informatics 9 (1) (2013) 427–438 (2013). `doi:10.1109/TII.2012.2219061`.

[51] B. Horling, V. Lesser, A survey of multi-agent organizational paradigms, The Knowledge Engineering Review 19 (4) (2004) 281–316 (2004). `doi:10.1017/S0269888905000317`.

[52] O. Boissier, J. F. Hübner, J. S. Sichman, Organization oriented programming: From closed to open organizations, in: International Workshop on Engineering Societies in the Agents World, Springer, 2006, pp. 86–105 (2006). `doi:10.1007/978-3-540-75524-1_5`.

[53] G. D. M. Serugendo, M.-P. Gleizes, A. Karageorgos, Self-organization in multi-agent systems, The Knowledge Engineering Review 20 (2) (2005) 165–189 (2005). `doi:10.1017/S0269888905000494`.

[54] A. Ferscha, Collective adaptive systems, in: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers, UbiComp/ISWC'15 Adjunct, ACM, New York, NY, USA, 2015, pp. 893–895 (2015). `doi:10.1145/2800835.2809508`.

[55] M. Mamei, F. Zambonelli, L. Leonardi, Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems, in: Engineering Societies in the Agents World III, Springer, 2003, pp. 68–81 (2003). `doi: 10.1007/3-540-39173-8_6`.

[56] D. Stovall, C. Julien, Resource discovery with evolving tuples, in: International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting, ESSPE, ACM, New York, NY, USA, 2007, pp. 1–10 (2007). `doi: 10.1145/1294904.1294905`.

[57] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, in: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013, Ch. 16, pp. 436–501, a longer version available at: `http://arxiv. org/abs/1202.5509` (2013). `doi:10.4018/978-1-4666-2092-6. ch016`.

[58] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, F. Zambonelli, A survey of autonomic communications, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1 (2) (2006) 223–259 (2006). `doi:10.1145/1186778. 1186782`.

[59] R. Menezes, R. Tolksdorf, Adaptiveness in Linda-based coordination models, in: Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering, Vol. 2977 of LNAI, Springer, 2004, pp. 212–232 (2004). `doi:10.1007/b95863`.

[60] A. Omicini, M. Viroli, Coordination models and languages: From parallel computing to self-organisation, The Knowledge Engineering Review 26 (1) (2011) 53–59 (2011). `doi:10.1017/S026988891000041X`.

[61] L. Clement, R. Nagpal, Self-assembly and self-repairing topologies, in: Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open, 2003 (2003).

[62] D. Yamins, A theory of local-to-global algorithms for one-dimensional spatial multi-agent systems, Ph.D. thesis, Harvard, Cambridge, MA, USA (2007).

[63] C. Lasser, J. Massar, J. Miney, L. Dayton, Starlisp Reference Manual, Thinking Machines Corporation, 1988 (1988).

[64] B. R. Engstrom, P. R. Cappello, The SDEF programming system, Journal of Parallel and Distributed Computing 7 (2) (1989) 201 – 231 (1989). `doi:10.1016/0743-7315(89)90018-X`.

[65] J. Beal, Programming an amorphous computational medium, in: Unconventional Programming Paradigms, Springer Berlin/Heidelberg, 2005, pp. 97–97 (2005). `doi:10.1007/11527800_10`.

[66] M. Viroli, F. Damiani, J. Beal, A calculus of computational fields, in: Advances in Service-Oriented and Cloud Computing, Vol. 393 of Communications in Computer and Information Science, Springer, 2013, pp. 114–128 (2013). `doi:10.1007/978-3-642-45364-9_11`.

[67] A. Church, A set of postulates for the foundation of logic, Annals of Mathematics 33 (2) (1932) 346–366 (1932). `doi:10.2307/1968337`.

[68] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (3) (2001) 396–450 (2001). `doi:10.1145/503502.503505`.

[69] F. Damiani, M. Viroli, J. Beal, A type-sound calculus of computational fields, Science of Computer Programming 117 (2016) 17 – 44 (2016). `doi:10.1016/j.scico.2015.11.005`.

[70] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Transactions on Modeling and Computer Simulation 28 (2) (2018) 1–28 (mar 2018). `doi:10.1145/3177774`.

[71] G. Audrito, F. Damiani, M. Viroli, R. Casadei, Run-time management of computation domains in field calculus, in: International Workshops on Foundations and Applications of Self* Systems (FAS*W), IEEE, 2016, pp. 192–197 (2016). `doi:10.1109/FAS-W.2016.50`.

[72] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Symposium on Principles of Programming Languages (POPL), ACM, 1982, pp. 207–212 (1982). `doi:10.1145/582153.582176`.

[73] R. Milner, The definition of standard ML: revised, MIT press, 1997 (1997).

[74] G. Audrito, J. Beal, F. Damiani, M. Viroli, Space-time universality of field calculus, in: International Conference on Coordination Models and Languages (COORDINATION), Vol. 10852 of Lecture Notes in Computer Science, Springer, 2018, pp. 1–20 (2018). `doi:10.1007/978-3-319-92408-3_1`.

[75] G. Winskel, Event structures, in: W. Brauer, W. Reisig, G. Rozenberg (Eds.), Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Vol. 255 of Lecture Notes in Computer Science, Springer, 1986, pp. 325–392 (1986). `doi:10.1007/3-540-17906-2_31`.

[76] S. Dolev, Self-Stabilization, MIT Press, 2000 (2000).

[77] A. Lluch-Lafuente, M. Loreti, U. Montanari, Asynchronous distributed execution of fixpoint-based computational fields, Logical Methods in Computer Science 13 (1) (2017). `doi:10.23638/LMCS-13(1:13)2017`.

[78] A. Lluch-Lafuente, M. Loreti, U. Montanari, A fixpoint-based calculus for graph-shaped computational fields, in: 17th International Conference on Coordination Models and Languages (COORDINATION), 2015, pp. 101–116 (2015). `doi:10.1007/978-3-319-19282-6_7`.

[79] F. Damiani, M. Viroli, Type-based self-stabilisation for computational fields, Logical Methods in Computer Science 11 (4) (2015). `doi:10.2168/LMCS-11(4:21)2015`.

[80] M. Viroli, F. Damiani, A calculus of self-stabilising computational fields, in: 16th International Conference on Coordination Models and Languages (COORDINATION), Vol. 8459 of Lecture Notes in Computer Science, Springer, 2014, pp. 163–178 (2014). `doi:10.1007/978-3-662-43376-8_11`.

[81] M. Viroli, J. Beal, F. Damiani, D. Pianini, Efficient engineering of complex self-organising systems by self-stabilising fields, in: 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2015, pp. 81–90 (September 2015). `doi:10.1109/SASO.2015.16`.

[82] J. Beal, M. Viroli, D. Pianini, F. Damiani, Self-adaptation to device distribution in the Internet of Things, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 12 (3) (2017) 12 (2017). `doi:10.1145/3105758`.

[83] G. Audrito, F. Damiani, M. Viroli, E. Bini, Distributed real-time shortest-paths computations with the field calculus, in: 2018 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2018, pp. 23–34 (dec 2018). `doi:10.1109/rtss.2018.00013`.

[84] R. Bakhshi, L. Cloth, W. Fokkink, B. R. Haverkort, Mean-field framework for performance evaluation of pushpull gossip protocols, Performance Evaluation 68 (2) (2011) 157 – 179, advances in Quantitative Evaluation of Systems (2011). `doi:10.1016/j.peva.2010.08.025`.

[85] F. Damiani, M. Viroli, D. Pianini, J. Beal, Code mobility meets self-organisation: A higher-order calculus of computational fields, in: Formal Techniques for Distributed Objects, Components, and Systems, Vol. 9039 of Lecture Notes in Computer Science, Springer, 2015, pp. 113–128 (2015). `doi:10.1007/978-3-319-19195-9_8`.

[86] D. Pianini, M. Viroli, J. Beal, Protelis: practical aggregate programming, in: Symposium on Applied Computing, ACM, 2015, pp. 1846–1853 (2015). `doi:10.1145/2695664.2695913`.

[87] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with ALCHEMIST, J. Simulation 7 (3) (2013) 202–215 (2013). `doi:10.1057/jos.2012.27`.

[88] D. G. Bell, F. Kuehnel, C. Maxwell, R. Kim, K. Kasraie, T. Gaskins, P. Hogan, J. Coughlan, NASA world wind: Opensource GIS for mission operations, in: Aerospace Conference, IEEE, 2007 (2007). `doi:10.1109/aero.2007.352954`.

[89] S. S. Clark, J. Beal, P. Pal, Distributed recovery for enterprise services, in: 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2015, pp. 111–120 (2015). `doi:10.1109/SASO.2015.19`.

[90] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, 2E, Packt Publishing, 2016 (2016).

[91] M. Francia, D. Pianini, J. Beal, M. Viroli, Towards a foundational API for resilient distributed systems design, in: International Workshops on Foundations and Applications of Self* Systems (FAS*W), IEEE, 2017 (2017). `doi:10.1109/fas-w.2017.116`.

[92] J. Beal, M. Viroli, Building blocks for aggregate programming of self-organising applications, in: 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2014, pp. 8–13 (2014). `doi:10.1109/SASOW.2014.6`.

[93] D. Pianini, J. Beal, M. Viroli, Improving gossip dynamics through overlapping replicates, in: 18th International Conference on Coordination Models and Languages (COORDINATION), Springer, 2016, pp. 192–207 (2016). `doi:10.1007/978-3-319-39519-7_12`.

[94] R. Casadei, M. Viroli, Towards aggregate programming in Scala, in: 1st Workshop on Programming Models and Languages for Distributed Computing, ACM, 2016, p. 5 (2016). `doi:10.1145/2957319.2957372`.

[95] M. Odersky, T. Rompf, Unifying functional and object-oriented programming with Scala, Communications of ACM 57 (4) (2014) 76–86 (2014). `doi:10.1145/2591013`.

[96] R. Casadei, D. Pianini, M. Viroli, Simulating large-scale aggregate MASs with Alchemist and Scala, in: Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2016, pp. 1495–1504 (2016). `doi:10.15439/2016F407`.

[97] R. Casadei, M. Viroli, Programming actor-based collective adaptive systems, in: Programming with Actors - State-of-the-Art and Research Perspectives, Vol. 10789 of Lecture Notes in Computer Science, Springer, 2018 (2018). `doi:10.1007/978-3-030-00302-9_4`.

[98] M. Viroli, R. Casadei, D. Pianini, On execution platforms for large-scale aggregate computing, in: International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, ACM, 2016, pp. 1321–1326 (2016). `doi:10.1145/2968219.2979129`.

[99] R. Roestenburg, R. Bakker, R. Williams, Akka in Action, 1st Edition, Manning Publications Co., Greenwich, CT, USA, 2015 (2015).

[100] Apache Kafka, retrieved March 13-th 2019, `https://kafka.apache.org` (2019).

[101] Apache Spark, retrieved March 13-th 2019, `https://spark.apache.org` (2019).

[102] B. C. Oliveira, A. Moors, M. Odersky, Type classes as objects and implicits, ACM Sigplan Notices 45 (10) (2010) 341–360 (2010). `doi:10.1145/1932682.1869489`.

[103] H. Curry, R. Feys, Combinatory logic, North-Holland, 1958 (1958).

[104] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, J. L. Arcos, Description and composition of bio-inspired design patterns: a complete overview, Natural Computing 12 (1) (2013) 43–67 (2013). `doi:10.1007/s11047-012-9324-y`.

[105] J. Beal, J. Bachrach, D. Vickery, M. Tobenkin, Fast self-healing gradients, in: Symposium on Applied computing, ACM, 2008, pp. 1969–1975 (2008). `doi:10.1145/1363686.1364163`.

[106] J. Beal, Flexible self-healing gradients, in: Symposium on Applied Computing, ACM, 2009, pp. 1197–1201 (2009).

[107] G. Audrito, F. Damiani, M. Viroli, Optimally-self-healing distributed gradient structures through bounded information speed, in: International Conference on Coordination Models and Languages (COORDINATION), Vol. 10319 of Lecture Notes in Computer Science, Springer, 2017, pp. 59–77 (2017). `doi:10.1007/978-3-319-59746-1_4`.

[108] G. Audrito, F. Damiani, M. Viroli, Optimal single-path information propagation in gradient-based algorithms, Sci. Comput. Program. 166 (2018) 146–166 (2018). `doi:10.1016/j.scico.2018.06.002`.

[109] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Compositional blocks for optimal self-healing gradients, in: 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2017, pp. 91–100 (2017). `doi:10.1109/SASO.2017.18`.

[110] J. Beal, Trading accuracy for speed in approximate consensus, The Knowledge Engineering Review 31 (4) (2016) 325–342 (2016). `doi:10.1017/S0269888916000175`.

[111] G. Audrito, F. Damiani, M. Viroli, Aggregate graph statistics, in: D. Pianini, G. Salvaneschi (Eds.), Proceedings of the First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Vol. 264 of EPTCS, 2017, pp. 18–22 (2017). `doi:10.4204/EPTCS.264.2.`

[112] G. Audrito, S. Bergamini, Resilient blocks for summarising distributed data, in: D. Pianini, G. Salvaneschi (Eds.), Proceedings of the First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Vol. 264 of EPTCS, 2017, pp. 23–26 (2017). `doi:10.4204/ EPTCS.264.3.`

[113] G. Audrito, S. Bergamini, F. Damiani, M. Viroli, Effective collective summarisation of distributed data in mobile multi-agent systems, in: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, 2019, pp. 1618–1626 (2019).
URL `http://dl.acm.org/citation.cfm?id=3331882`

[114] M. Viroli, J. Beal, Resiliency with aggregate computing: State of the art and roadmap, in: Workshop on FORmal methods for the quantitative Evaluation of Collective Adaptive SysTems (FORECAST), 2016 (2016). `doi:10.4204/EPTCS.217.3.`

[115] N. Lynch, Distributed Algorithms, Morgan Kaufmann, San Francisco, USA, 1996 (1996).

[116] D. Shevitz, B. Paden, Lyapunov stability theory of nonsmooth systems, IEEE Transactions on automatic control 39 (9) (1994) 1910–1914 (1994). `doi:10.1109/CDC.1993.325114.`

[117] M. Arcak, Passivity as a design tool for group coordination, IEEE Transactions on Automatic Control 52 (8) (2007) 1380–1390 (2007). `doi: 10.1109/TAC.2007.902733.`

[118] T. Hatanaka, N. Chopra, M. Fujita, M. W. Spong, Passivity-based control and estimation in networked robotics, Springer, 2015 (2015). `doi:10. 1007/978-3-319-15171-7.`

[119] B. D. Anderson, The small-gain theorem, the passivity theorem and their equivalence, Journal of the Franklin Institute 293 (2) (1972) 105–115 (1972). `doi:10.1016/0016-0032(72)90150-0Get.`

[120] T. H. Summers, C. Yu, S. Dasgupta, B. D. Anderson, Control of minimally persistent leader-remote-follower and coleader formations in the plane, IEEE Transactions on Automatic Control 56 (12) (2011) 2778–2792 (2011). `doi:10.1109/TAC.2011.2146890.`

[121] L. Krick, M. E. Broucke, B. A. Francis, Stabilisation of infinitesimally rigid formations of multi-robot networks, International Journal of control 82 (3) (2009) 423–439 (2009). `doi:10.1080/00207170802108441.`

[122] R. Olfati-Saber, J. A. Fax, R. M. Murray, Consensus and cooperation in networked multi-agent systems, Proceedings of the IEEE 95 (1) (2007) 215–233 (2007). `doi:10.1109/JPROC.2006.887293`.

[123] Z.-P. Jiang, I. M. Mareels, Y. Wang, A Lyapunov formulation of the nonlinear small-gain theorem for interconnected ISS systems, Automatica 32 (8) (1996) 1211–1215 (1996). `doi:10.1016/0005-1098(96) 00051-9`.

[124] M. Fu, S. Dasgupta, Y. C. Soh, Integral quadratic constraint approach vs. multiplier approach, Automatica 41 (2) (2005) 281–287 (2005). `doi: 10.1016/j.automatica.2004.10.005`.

[125] M. Fu, S. Dasgupta, Parametric Lyapunov functions for uncertain systems: The multiplier approach, in: Advances in linear matrix inequality methods in control, SIAM, 2000, pp. 95–108 (2000).

[126] A. Kumar, J. Beal, S. Dasgupta, R. Mudumbai, Toward predicting distributed systems dynamics, in: International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), IEEE, 2015, pp. 68–73 (2015). `doi:10.1109/SASOW.2015.16`.

[127] Y. Mo, J. Beal, S. Dasgupta, An aggregate computing approach to self-stabilizing leader election, in: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), IEEE, 2018, pp. 112–117 (2018). `doi:10.1109/fas-w.2018.00034`.

[128] S. Dasgupta, J. Beal, A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm, in: 55th IEEE Conference on Decision and Control (CDC), IEEE, 2016, pp. 7282–7287 (2016). `doi:10.1109/ CDC.2016.7799393`.

[129] Y. Mo, S. Dasgupta, J. Beal, Robust stability of spreading blocks in aggregate computing, in: 2018 IEEE Conference on Decision and Control (CDC), IEEE, 2018, pp. 6007–6012 (dec 2018). `doi:10.1109/cdc. 2018.8618735`.

[130] Y. Mo, J. Beal, S. Dasgupta, Error in self-stabilizing spanning-tree estimation of collective state, in: International Workshops on Foundations and Applications of Self* Systems (FAS*W), IEEE, 2017, pp. 1–6 (2017). `doi:10.1109/FAS-W.2017.112`.

[131] G. Audrito, F. Damiani, V. Stolz, M. Viroli, On distributed runtime verification by aggregate computing, in: Post-proceedings of the 3rd International Workshop on Verification of Objects at Runtime Execution (VOR-TEX 2018), EPTCS, to appear.

[132] G. Audrito, J. Beal, F. Damiani, D. Pianini, M. Viroli, The share operator for field-based coordination, in: H. R. Nielson, E. Tuosto (Eds.), 21st

International Conference on Coordination Models and Languages (COOR-DINATION), Vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 54–71 (2019). `doi:10.1007/978-3-030-22397-7_4`.

[133] L. Bortolussi, J. Hillston, D. Latella, M. Massink, Continuous approximation of collective system behaviour: A tutorial, Performance Evaluation 70 (5) (2013) 317–349 (2013). `doi:10.1016/j.peva.2013.01.001`.

[134] V. Ciancia, S. Gilmore, G. Grilletti, D. Latella, M. Loreti, M. Massink, Spatio-temporal model checking of vehicular movement in public transport systems, International Journal on Software Tools for Technology Transfer (2018) 1–23 (2018). `doi:10.1007/s10009-018-0483-8`.

[135] J. Bachrach, J. Beal, J. McLurkin, Composable continuous-space programs for robotic swarms, Neural Computing and Applications 19 (6) (2010) 825–847 (2010). `doi:10.1007/s00521-010-0382-8`.

[136] J. Beal, A tactical command approach to human control of vehicle swarms., in: AAAI Fall Symposium: Human Control of Bioinspired Swarms, 2012 (2012).

[137] J. Beal, Superdiffusive dispersion and mixing of swarms, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 10 (2) (2015) Article 10 (2015). `doi:10.1145/2700322`.

[138] M. Viroli, D. Pianini, A. Ricci, A. Croatti, Aggregate plans for multiagent systems, International Journal of Agent-Oriented Software Engineering 4 (5) (2017) 336–365 (2017). `doi:10.1504/IJAOSE.2017.087638`.

[139] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Aggregate processes in field calculus, in: H. R. Nielson, E. Tuosto (Eds.), 21st International Conference on Coordination Models and Languages (COOR-DINATION), Vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 200–217 (2019). `doi:10.1007/978-3-030-22397-7_12`.

[140] R. Casadei, M. Viroli, Collective abstractions and platforms for large-scale self-adaptive iot, in: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), IEEE, 2018, pp. 106–111 (2018). `doi:10.1109/fas-w.2018.00033`.

[141] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, A development approach for collective opportunistic edge-of-things services, Information Sciences 498 (2019) 154–169 (2019). `doi:10.1016/j.ins.2019.05.058`.

[142] D. Schmidt, Middleware for Real-time and Embedded Systems, Communications of the ACM 45 (6) (2002) 43–48 (2002). `doi:10.1145/508448.508472`.

[143] M. Iorgam, L. Feldman, R. Barton, M. J. Martin, N. Goren, C. Mahmoudi, Fog Computing Conceptual Model, Recommendations of the National Institute of Standards and Technology, NIST Special publication 500-325, 2018.

[144] Openfog consortium, retrieved January, 2019, `https://www.openfogconsortium.org` (2019).

[145] S. Hachem, V. Issareny, V. Mallet, A. Pathak, P. Bhatia, R. amd Raverdy, Urban Civics: An IoT middleware for democratizing crowdsensed data in smart societies, in: Research and Tech. for Society and Industry Leveraging a Better Tomorrow (RTSI), 2015, pp. 117–126 (2015). `doi:10.1109/RTSI.2015.7325081`.

[146] R. Casadei, D. Pianini, M. Viroli, A. Natali, Self-organising coordination regions: A pattern for edge computing, in: 21st International Conference on Coordination Models and Languages (COORDINATION), Lecture Notes in Computer Science, Springer, 2019, pp. 182–199 (2019). `doi:10.1007/978-3-030-22397-7_11`.

[147] J. Beal, M. Viroli, D. Pianini, F. Damiani, Self-adaptation to device distribution changes, in: 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016, Augsburg, Germany, September 12-16, 2016, 2016, pp. 60–69 (2016). `doi:10.1109/SASO.2016.12`.

[148] R. Casadei, A. Aldini, M. Viroli, Towards attack-resistant aggregate computing using trust mechanisms, Science of Computer Programming 167 (2018) 114–137 (2018). `doi:10.1016/j.scico.2018.07.006`.

[149] A. Mourad, M.-A. Laverdière, M. Debbabi, Security hardening of open source software, in: Proceedings of the 2006 International Conference on Privacy, Security and Trust Bridge the Gap Between PST Technologies and Business Services - PST 06, ACM Press, 2006 (2006). `doi:10.1145/1501434.1501486`.

[150] D. Pianini, G. Ciatto, R. Casadei, S. Mariani, M. Viroli, A. Omicini, Transparent protection of aggregate computations from byzantine behaviours via blockchain, in: Proceedings of the 4th EAI International Conference on Smart Objects and Technologies for Social Good, ACM, 2018, pp. 271–276 (2018). `doi:10.1145/3284869.3284870`.

[151] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem, ACM Transactions on Programming Languages and Systems (TOPLAS) 4 (3) (1982) 382–401 (1982). `doi:10.1145/357172.357176`.

[152] Z. Zheng, S. Xie, H. Dai, X. Chen, H. Wang, An overview of blockchain technology: Architecture, consensus, and future trends, in: 2017 IEEE international congress on big data (BigData congress), IEEE, 2017, pp. 557–564 (2017). `doi:10.1109/BigDataCongress.2017.85`.

[153] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium (USENIX Security 18), 2018 (2018).
URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[154] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, in: 40th IEEE Symposium on Security and Privacy (S&P'19), 2019 (2019).

[155] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors, in: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), IEEE, 2014 (jun 2014). doi:10.1109/isca.2014.6853210.

[156] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, K. Razavi, Throwhammer: Rowhammer attacks over the network and defenses, in: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, USENIX Association, Berkeley, CA, USA, 2018, pp. 213–225 (2018).
URL http://dl.acm.org/citation.cfm?id=3277355.3277377

[157] B. Penzenstadler, A. Raturi, D. Richardson, B. Tomlinson, Safety, security, now sustainability: The nonfunctional requirement for the 21st century, IEEE Software 31 (3) (2014) 40–47 (may 2014). doi:10.1109/ms.2014.22.

[158] J. Bachrach, J. Beal, Building spatial computers, Tech. Rep. CSAIL Tech Report 2007-017, MIT (March 2007).

[159] M. Bor, J. Vidler, U. Roedig, Lora for the internet of things, in: Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks, EWSN '16, Junction Publishing, USA, 2016, pp. 361–366 (2016).
URL http://dl.acm.org/citation.cfm?id=2893711.2893802

[160] D. Pianini, A. Elzanaty, A. Giorgetti, M. Chiani, Emerging distributed programming paradigm for cyber-physical systems over LoRaWANs, in: 2018 IEEE Globecom Workshops (GC Wkshps), 2018, pp. 1–6 (Dec 2018). doi:10.1109/GLOCOMW.2018.8644518.

[161] H. H. R. Sherazi, G. Piro, L. A. Grieco, G. Boggia, When renewable energy meets LoRa: A feasibility analysis on cable-less deployments, IEEE Internet of Things Journal (2018) 1–1 (2018). doi:10.1109/jiot.2018.2839359.

[162] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, Modelling and simulation of opportunistic IoT services with aggregate computing, Future Generation Computer Systems 91 (2019) 252–262 (feb 2019). `doi:10.1016/j.future.2018.09.005`.

[163] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, ACM, 2012, pp. 13–16 (2012). `doi: 10.1145/2342509.2342513`.

[164] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet of Things Journal 3 (5) (2016) 637–646 (2016). `doi:10.1109/JIOT.2016.2579198`.

[165] J. Beal, M. Viroli, Space–time programming, Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 373 (2046) (2015). `doi:10.1098/rsta.2014.0220`.

[166] H. V. D. Parunak, S. Brueckner, Concurrent modeling of alternative worlds with polyagents, in: Multi-Agent-Based Simulation VII, Springer, 2007, pp. 128–141 (2007). `doi:10.1007/978-3-540-76539-4_10`.

[167] J. Beal, K. Usbeck, J. Loyall, J. Metzler, Opportunistic sharing of airborne sensors, in: International Conference on Distributed Computing in Sensor Systems (DCOSS), IEEE, 2016, pp. 25–32 (2016). `doi: 10.1109/DCOSS.2016.43`.

[168] J. Beal, K. Usbeck, J. Loyall, M. Rowe, J. Metzler, Adaptive opportunistic airborne sensor sharing, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 13 (1) (2018) 6 (2018). `doi:10.1145/3179994`.

[169] J. Beal, M. Viroli, Aggregate programming: From foundations to applications, in: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems, Springer, 2016, pp. 233–260 (2016). `doi: 10.1007/978-3-319-34096-8_8`.