Aggregate Processes in Field Calculus

Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, Ferruccio Damiani

Aggregate Processes in Field Calculus*

Roberto Casadei[1][0000−0001−9149−949X], Mirko Viroli[1][0000−0003−2702−5702], Giorgio Audrito[2][0000−0002−2319−0375], Danilo Pianini[1][0000−0002−8392−5409], and Ferruccio Damiani[2][0000−0001−8109−1706]

[1] Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{roby.casadei, mirko.viroli, danilo.pianini}@unibo.it
[2] Università di Torino, Italy
{giorgio.audrito, ferruccio.damiani}@unito.it

Abstract. Engineering distributed applications and services in emerging and open computing scenarios like the Internet of Things, cyber-physical systems and pervasive computing, calls for identifying proper abstractions to smoothly capture collective behaviour, adaptivity, and dynamic injection and execution of concurrent distributed activities. Accordingly, we introduce a notion of "aggregate process" as a concurrent field computation whose execution and interactions are sustained by a dynamic team of devices, and whose spatial region can opportunistically vary over time. We formalise this notion by extending the Field Calculus with a new primitive construct, `spawn`, used to instantiate a set of field computations and regulate key aspects of their life-cycle. By virtue of an open-source implementation in the ScaFi framework, we show basic programming examples and benefits via two case studies of mobile ad-hoc networks and drone swarm scenarios, evaluated by simulation.

Keywords: aggregate processes · computational fields · distributed computing · collective coordination · dynamic ensembles · self-*.

1 Introduction

Emerging scenarios like pervasive computing, Internet of Things (IoT), cyber-physical systems (CPS) and edge computing, are leading towards a new reference computational *fabric* made of dense, large-scale networks of heterogeneous devices. New opportunities for developing software services naturally arise that fully leverage the pervasive availability of sensing, actuation, storage, computational power and networking. To help unveiling the true potentials of such digitally empowered ecosystems, proper abstractions and development techniques are needed to smoothly express *collective* coordination and computation activities that can be transparently executed on opportunistic formations of devices [10].

* This work has been partially supported by Ateneo/CSP project "AP: Aggregate Programming" (`http://ap-project.di.unito.it/`).

In such contexts, computational events might trigger multiple distributed activities that are highly contextual and hence fundamentally related to their space-time situation and physical environment. Openness and dynamism, then, require such activities to be dependable, self-adaptive and self-organising in order to maintain coherence and functionality across unpredictable and inevitable context changes and adversary events, and to opportunistically activate wherever and whenever their existence conditions hold—whether they are by-design or emergent. For instance, for collaborative smartphone-based applications in a smart city, such activities may include: a gossip process by which people in a plaza share comments, a guidance process to make a group of friends gather in a convenient point, a dispersal process for people creating bloat, a process to advertise one's presence to nearby users for the next minute, a process to provide crowd-aware directions towards a point of interest, and so on [5,38,25,31,8].

According to this vision, we present the concept of *aggregate process*, denoting a distributed computation sustained by a dynamic aggregation of devices—hence using the term *aggregate* with the meaning of "pertaining to a collective", i.e., in the sense of [5,35]. This abstraction can be useful to model transient collective activities, which may concurrently span and overlap over the fabric created by a mobile, large-scale deployment of devices; it is aimed to capture: *(i) aggregate stance*, to promote pervasive adaptation, by abstracting the individual device and seamlessly regulating the behaviour of an ensemble across scales, density, and heterogeneity; *(ii) dynamicity and context-orientation*, to conveniently support the implementation of dynamic, distributed, spatio-temporal activities where locality and context play a major role, and continuous change is the norm; *(iii) intrinsic resiliency*, to specify and execute collective (inter-)actions independently of large classes of environmental dynamics and faults. This notion, hence, fosters a broader view of programming smart distributed environments like sorts of *distributed virtual machines* for aggregate processes, supporting the dynamic injection and execution of collective computations, their diffusion over an opportunistically selected region of space-time, and their inherent self-adaptation to changes and faults by full abstraction over individual behaviours of devices.

To formally capture the features of aggregate processes, and experiment with mechanisms to handle their life-cycle (process creation, disposal, logic and interaction), we adopt as basis framework the *field calculus* [4,35]—a coordination model based on the notion of *(computational) field* (a time-evolving distributed structure mapping devices to computational values) where coordination policies are declaratively and compositionally expressed as pure functions from fields to fields. As key contribution, aggregate processes are supported in the field calculus by a new primitive construct, `spawn`, yielding a field that, across space and time, combines several independent but interacting "computational bubbles" (process instances). Programming constructs to work with aggregate processes are implemented in ScaFi [9,11] (`https://github.com/scafi/scafi`), a Scala-based incarnation of field calculus: this is used to showcase the expressiveness of the notion and to empirically evaluate the proposed abstraction through simulation of two paradigmatic case studies of mobile ad-hoc networks and drone swarms.

The remainder of this paper is organised as follows. Section 2 presents field calculus and its extension to support aggregate processes. Section 3 describes implementation in ScaFi along with examples and programming techniques. Section 4 provides evaluation of aggregate processes through synthetic experiments. Section 5 concludes the paper with discussion of related and future work.

2 Founding Aggregate Processes by the Field Calculus

Founding the notion of aggregate processes requires a coordination model with the power to declaratively express complex spatio-temporal behaviour possibly involving large sets of networked devices. Among the various frameworks enabling such a "macro-programming" paradigm, reviewed in Section 5, we consider the field calculus [4] (FC). This is a minimal functional language that captures the foundational mechanisms for compositionally expressing the emergent behaviour of a collective system by a global perspective. It provides constructs to represent and manipulate *(computational) fields*, i.e., distributed and time-evolving data structures that map device identities to computational values.

Arguably, FC represents a natural basis for technically developing a notion of aggregate process—which in fact somewhat emerged from technical issues about field computations. Indeed, FC enables an *aggregate stance* to programming: field computations target a collective of devices as a whole, and the field semantics formally provides a bridge from global behaviour to local activity of individual devices. *Dynamicity and context-orientation* are also directly supported: a system is modelled as a logical network of devices connected through a neighbouring relationship; devices can sample their portion of the environment and communicate with neighbours to infer/propagate context and react to changes in their surroundings. Moreover, the model also provides *inherent resiliency*, by abstracting from networking issues and adopting an execution model where computations are "continuously" re-evaluated in order to sustain field evolution in spite of individual failures and outages.

In this section, we briefly introduce FC (Section 2.1—the reader interested in full technical details should refer to [4]); then, we motivate the need for specific mechanisms to support a true notion of "process" (Section 2.2); finally, we conclude with the formalisation of a new primitive construct `spawn` (Section 2.3), responsible for managing (i.e., activating, executing, closing) a dynamic number of field computations (i.e., process instances).

2.1 Overview of Field Calculus

Figure 1 (first frame) presents the syntax and device semantics of FC, where the grey-boxed parts correspond to the new `spawn` construct and will be explained in Section 2.3. Following [24], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by "•": e.g., for expressions, we let $\overline{\mathtt{e}}$ range over sequences of expressions, written $\mathtt{e}_1$, $\mathtt{e}_2$, ... $\mathtt{e}_n$ ($n \geq 0$). A program `P` consists of a sequence of function declarations and of a main expression `e`. A function declaration `F` defines a (possibly recursive) function. It consists of

Syntax:

$\texttt{P} ::= \overline{\texttt{F}}\ \texttt{e}$ program | $\texttt{F} ::= \texttt{def d}(\overline{\texttt{x}})\ \{\texttt{e}\}$ function declaration

$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{rep(e)}\{(\texttt{x}) \texttt{ => e}\} \mid \texttt{nbr}\{\texttt{e}\} \mid \texttt{spawn(e, e, e)}$ expression

$\texttt{v} ::= \phi \mid \ell$ value | $\phi ::= \overline{\delta} \mapsto \overline{\ell}$ neighbouring field value

$\ell ::= \texttt{f} \mid \texttt{c}(\overline{\ell})$ local value | $\texttt{f} ::= \texttt{b} \mid \texttt{d} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}$ function value

Value-trees and value-tree environments:

$\theta ::= \texttt{v} \mid \texttt{v}\langle\overline{\theta}\rangle \mid \overline{\texttt{v}} \mapsto \overline{\theta}$ value-tree

$\Theta ::= \overline{\delta} \mapsto \overline{\theta}$ value-tree environment

Auxiliary functions:

$args((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \overline{\texttt{x}}$ | $body((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \texttt{e}$ | $name((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{e}) = \tau$

$args(\texttt{d}) = \overline{\texttt{x}}$ if $\texttt{def d}(\overline{\texttt{x}})\{\texttt{e}\}$ | $body(\texttt{d}) = \texttt{e}$ if $\texttt{def d}(\overline{\texttt{x}})\{\texttt{e}\}$ | $name(\texttt{d}) = \texttt{d}$

$\rho(\texttt{v}\langle\overline{\theta}\rangle) = \texttt{v}$ | | $name(\texttt{b}) = \texttt{b}$

$\pi_i(\texttt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_i$ | if $1 \leq i \leq n$ | else $\bullet$

$\pi^{\texttt{f}}(\texttt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_n$ | if $name(\rho(\theta_1)) = name(\texttt{f})$ | else $\bullet$

$\pi^k(\overline{\texttt{v}} \mapsto \overline{\theta}) = \theta_i$ | s.t. $\texttt{v}_i = k$ if it exists | else $\bullet$

$F(\theta) = \texttt{v}\langle\overline{\theta}\rangle$ | if $\theta = \texttt{pair}(\texttt{v}, \texttt{True})\langle\overline{\theta}\rangle$ | else $\bullet$

For $aux \in \rho, \pi_i, \pi^{\texttt{f}}, \pi^k, F$:

$$\begin{cases} aux(\bullet) = \bullet & \\ aux(\delta \mapsto \theta, \Theta) = aux(\Theta) & \text{if } aux(\theta) = \bullet \\ aux(\delta \mapsto \theta, \Theta) = \delta \mapsto aux(\theta), aux(\Theta) & \text{if } aux(\theta) \neq \bullet \end{cases}$$

Rules for expression evaluation: $\boxed{\delta; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta}$

$$\text{[E-APP]}\ \frac{\begin{array}{c}\delta; \pi_1(\Theta); \sigma \vdash \texttt{e} \Downarrow \theta \qquad \delta; \pi_{i+1}(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta_i \text{ for } i = 1 \ldots n \qquad \texttt{f} = \rho(\theta) \\ \theta' = (\!|\texttt{f}|\!)^{\pi^{\texttt{f}}(\Theta)}_{\delta,\sigma}(\rho(\overline{\theta})) \text{ if f built-in else } \delta; \pi^{\texttt{f}}(\Theta); \sigma \vdash body(\texttt{f})[args(\texttt{f}) := \rho(\overline{\theta})] \Downarrow \theta'\end{array}}{\delta; \Theta; \sigma \vdash \texttt{e}(\overline{\texttt{e}}) \Downarrow \rho(\theta')\langle\theta, \overline{\theta}, \theta'\rangle}$$

$$\text{[E-LOC]}\ \frac{}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle} \qquad \text{[E-FLD]}\ \frac{\phi' = \phi|_{\mathbf{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle} \qquad \text{[E-NBR]}\ \frac{\Theta_1 = \pi_1(\Theta) \quad \delta; \Theta_1; \sigma \vdash \texttt{e} \Downarrow \theta_1}{\delta; \Theta; \sigma \vdash \texttt{nbr}\{\texttt{e}\} \Downarrow \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]\langle\theta_1\rangle}$$

$$\text{[E-REP]}\ \frac{\begin{array}{ll}\delta; \pi_1(\Theta); \sigma \vdash \texttt{e}_1 \Downarrow \theta_1 & \ell_1 = \rho(\theta_1) \\ \delta; \pi_2(\Theta); \sigma \vdash \texttt{e}_2[\texttt{x} := \ell_0] \Downarrow \theta_2 & \ell_2 = \rho(\theta_2)\end{array} \quad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \ell_1 & \text{otherwise} \end{cases}}{\delta; \Theta; \sigma \vdash \texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{ => e}_2\} \Downarrow \ell_2\langle\theta_1, \theta_2\rangle}$$

$$\text{[E-SPAWN]}\ \frac{\begin{array}{ll}\delta; \pi_i(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta^i & \text{for } i \in 1, 2, 3 \\ k_1, \ldots, k_n = \rho(\theta^2) \cup \bigcup\{\mathbf{dom}(\pi_4(\Theta(\delta'))) & \text{for } \delta' \in \mathbf{dom}(\Theta)\} \\ \delta; \pi^{k_i}(\pi_4(\Theta)); \sigma \vdash \rho(\theta_1)(k_i, \rho(\theta_3)) \Downarrow \theta_i & \text{for } i \in 1, \ldots, n\end{array}}{\delta; \Theta; \sigma \vdash \texttt{spawn}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3) \Downarrow F(\overline{k} \mapsto \rho(\overline{\theta}))\langle\theta^1, \theta^2, \theta^3, F(\overline{k} \mapsto \overline{\theta})\rangle}$$

Fig. 1: Syntax and device semantics for the field calculus (extended part in grey).

the name of the function d, of $n \geq 0$ variable names $\overline{\texttt{x}}$ representing the formal parameters, and of an expression e representing the body of the function. Expressions e are the main entities of the calculus, and will evaluate to a whole field, understood at the macro-level as a space/time-wide data structure, mapping *computational events* (i.e., when and where a device executes a computation) to values: the set of such computational events is called *field domain*. Expressions include rather standard functional constructs, like: *variables* x, used as function formal parameters; *values* v (described below); and *anonymous function expres-*

sions $(\overline{\texttt{x}})$ $\overset{\tau}{\texttt{=>}}$ $\texttt{e}$, where $\overline{\texttt{x}}$ are the formal parameters, $\texttt{e}$ is the body and τ is a *tag*[3]. A value can be either a *neighbouring value* ϕ or a *local value* ℓ. Technically, a neighbouring value is a mapping from device identifiers (corresponding to a device's neighbourhood including the device itself) to local values, while a local value can be: *(i)* a *data value* $\texttt{c}(\overline{\ell})$, consisting of a data-constructor applied to local value arguments (`true`, `false`, `0`, `1`, `pair(1,2)` and so on); or *(ii)* a *function value* `f`, consisting of either a declared function name `d`, a closed anonymous function, or a built-in function name `b` always working locally—used to denote usual mathematical/logical operators (e.g., `+`, `-`, `or`), 0-ary sensors (e.g., `temperature`, `pressure`, `sns`), or functions to turn neighbouring values to local values (e.g. minimisation of values by `minHood`, or minimisation excluding the device itself by `minHoodPlus`).

We model the computation of a device at each event by a big-step operational semantics where the result of evaluation is a *value-tree (vtree)* θ, i.e., an ordered tree of values that tracks the results of all evaluated subexpressions. The vtrees produced by an evaluation are made available to neighbours (including the device itself) for their forthcoming event through a broadcast. The evaluation of an expression at a given time in a device is thus performed "against" the recently-received vtrees of neighbours, as collected into a *vtree environment* Θ, mapping device identifiers to vtrees. The syntax of vtrees and vtree environments is given in Figure 1 (second frame). The operational semantics judgement is of the form $\delta; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta$, to be read "expression `e` evaluates to vtree θ on device δ w.r.t. the vtree environment Θ and sensor state σ", where: *(i)* δ is the identifier of the current device; *(ii)* Θ is the neighbouring field of the vtrees produced by the most recent evaluation of (an expression corresponding to) `e` on δ's neighbours; *(iii)* σ is a data structure containing enough sensor information to allow each non-pure built-in to be computed; *(iv)* `e` is an expression; *(v)* the vtree θ represents the values computed for all the expressions encountered during the evaluation of `e`—in particular $\rho(\theta)$ is the resulting value of `e`.

Expressions include also constructs that are tailored to field computations. A *function call* $\texttt{e}_f(\overline{\texttt{e}})$ adapts the standard call notion with the fact that $\texttt{e}_f$ is a field and hence could evaluate to different functions at different events, in which case it provides an advanced branching mechanism: the domain is partitioned in regions by the identity of such functions (determined by tag τ for anonymous functions, and by name for other functions), function application in each region applies the single function being there, and finally juxtaposition is applied to all regions. The function call mechanism is used to implement conventional branching, which also splits the domain of computation into two non-overlapping regions defined by where `e` evaluates to `true` or `false` ($\texttt{e}_1$ is executed in isolation in the former, $\texttt{e}_2$ is in the latter, and the juxtaposition of the two sub-fields defines the overall result). Namely, $\texttt{if(e)\{e}_1\texttt{\}\{e}_2\texttt{\}}$ is syntactic sugar for $\texttt{mux(e,}() \overset{\tau_1}{\texttt{=>}} \texttt{e}_1, () \overset{\tau_2}{\texttt{=>}} \texttt{e}_2)()$, where

[3] Tags τ do not appear in source programs: when the evaluation starts, each anonymous function expression $(\overline{\texttt{x}})$ `=>` `e` occurring in the program is converted into a tagged anonymous function expression by giving it a tag that is uniquely determined by its syntactical representation—see [4] for a detailed explanation.

the built-in `mux` is simply a multiplexer (it takes three arguments, evaluates all of them, and returns the second if the first has value `true` ot the third otherwise). A `rep`-*expression* `rep`(e_0){(`x`) => e_1} models fields evolving over time: the result field is initially e_0, and iteratively at each device function (`x`) => e_1 is applied to obtain the value at an event based on the value at previous one—e.g., `rep`(0){(`x`) => `x` + 1} is the field that counts the number of occurred events at each device. Finally, a `nbr`-*expression* `nbr{e}` is used to model device-to-neighbourhood interaction: at each device, it gives a local map from neighbours to values (a so-called *neighbouring value*) filled with the most recent results of evaluating `e` at each neighbour.

A key aspect of how the operational semantics is developed is called "alignment" [4,3]: to implement coherent sharing of values, an instance of operator `nbr` (say it is localised in position p of the vtree), is such that it gathers values from neighbours by retrieving them in the same position (p) of all vtrees contained in Θ. This is the cornerstone technique to support a declarative and compositional specification of interactions, and hence, of global level coordination.

2.2 On "Multiple Alignments"

Conceptually, and technically, FC is used to specify a "single field computation" working on the entire available domain. As a paradigmatic example, consider a `gradient` [34,2,25], namely, a field of hop-by-hop distances based on local estimates `metric` (a field of neighbouring real values) from the closest node in *source* (a field of boolean values):

```
def gradient(source, metric) {
  rep(infinity)(distance =>
     mux(source, 0, minHoodPlus( nbr{distance} + metric ))) }

def limitedGradient(source, metric, area) {
  if (area) {gradient(source, metric)} {infinity} }
```

If `sns` is a sensor giving `true` only at a device s (and `false` everywhere else) and `nbrRange` is a sensor giving local estimate distances from neighbours (as a range detector would support), then the main expression `gradient(sns,nbrRange)` gives a field stabilising to a situation where each device is mapped to its (hop-by-hop, nearest) distance to s [4,34,2,25,16]. If multiple devices are sources, estimated distance considers the nearest source.

There are mechanisms in FC to tweak this "single field computation" model. First of all, one could realise two computations by a field of pairs of values, say `pair(v1,v2)`: e.g., expression `pair(gradient(sns1,nbrRange), gradient(sns2,nbrRange))` would actually generate two completely independent gradient computations. The same approach is applicable to realise an arbitrary number of computations, but this practically works only if the number of such computations is small, known, and uniform across space and time, otherwise, FC has no mechanism to capture the abstraction of "aligned iteration" over a collection of values conceptually belonging to different computations.

A second key aspect involves the ability to restrict the domain of a computation. It is true that, by branching, one can prevent evaluation of some subexpressions—e.g., in function `limitedGradient`, if `area` is a boolean field giving `true` to a small subdomain, then computation of `gradient` is limited there. However, this approach has limitations as well: if one wants to limit a gradient to span the ball-like area where distances from the source are smaller than a given value, hence setting `area` to "`gradient(source,metric) < range`", there would be no direct way of avoiding computation of `gradient` outside that limited ball, because the decision on whether an event is inside or outside the ball has to be reconsidered everywhere and everytime.

So, technically, in FC there are no constructs to directly model, e.g., a reusable function that turns a field of boolean `sources` into a collection of independent gradients, one per source: that would require to create a field of lists of reals, of arbitrary size across space-time, but crucially this would not correctly support alignment. More generally, and although being universal [1], FC falls short in expressing the situation in which a field computation is composed of a set of subcomputations that is dynamic in the sense that has changing size over space and time. But this is precisely what is needed to support aggregate processes.

2.3 The `spawn` Construct Extension

We formalise our notion of *aggregate process* by extending FC with a `spawn` mechanism essentially carrying on a multiple aligned execution of concurrent computations, managing their life-cycle (i.e., activation, execution, disposal). Syntactically (see Figure 1), this is formed by a `spawn`-*expression* $\texttt{spawn}(\texttt{e}_b, \texttt{e}_k, \texttt{e}_i)$, modelling a collection of aggregate processes. Expression $\texttt{e}_b$ models process behaviour: it is a function (of informal type $k \to a \to \langle v, bool \rangle$) taking a process key (i.e., an identifier) and an input argument, and returning a pair of an output value and a boolean stating whether the process should be maintained alive or not. Expression $\texttt{e}_k$ defines a field of process keysets to add at each location (device); and $\texttt{e}_i$ is the input field to feed processes. The result of `spawn` is a field of maps from process keys to values. As a result, we can precisely define an *aggregate process with key* k as the projection to k of the field of maps resulting from `spawn`, that is, the computational field associating each event to the value corresponding to k at that event—as this may simply be absent at an event, aggregate processes are to be considered partial fields over the whole domain.

The semantic details of `spawn` are presented in grey in Figure 1. On the second frame, we allow to express vtrees also as $\overline{\texttt{v}} \mapsto \overline{\theta}$, i.e., as a map from keys to vtrees. On the third frame, we define auxiliary functions ρ, π_i, π^k for extracting from a vtree respectively: its root value, an ordered subtree by its index i, and an unordered subtree by its key k. It also defines a *filtering* function F which selects vtrees whose root is a pair $\texttt{pair}(\texttt{v}, \texttt{True})$, collapsing the root into `v`. All of these functions can be extended to maps (see *aux*), which are intended to be unordered vtree nodes for F, and vtree environments for ρ, π_i and π^k.

Finally, in fourth frame, we define the behaviour of construct `spawn`, formalised by the big-step operational semantics rule [E-SPAWN]: the sub-expressions

$\texttt{e}_1$, $\texttt{e}_2$ and $\texttt{e}_3$ are evaluated and their results stored in vtrees θ^1, θ^2, θ^3 forming the first branches of the final result. Then, a list of *process keys* $\overline{k}$ is computed by adjoining *(i)* the keys currently present in the result $\rho(\theta^2)$ of $\texttt{e}_2$; *(ii)* the keys that any neighbour δ' broadcast in their last unordered sub-branch $\pi_4(\Theta(\delta'))$. To realise "multiple alignment", for each key k_i, the process $\rho(\theta^1)$ resulting from evaluation of $\texttt{e}_1$ is applied to k_i and the result $\rho(\theta^3)$ of $\texttt{e}_3$, producing θ_i as a result. The map $\overline{k} \mapsto \overline{\theta}$ is then filtered by F, discarding evaluations resulting in a $\texttt{pair}(\texttt{v}, \texttt{False})$, before being made available to neighbours. The same results $F(\overline{k} \mapsto \rho(\overline{\theta}))$ are also returned as the root of the resulting vtree.

3 Programming with Aggregate Processes

In this section, we show how the `spawn` construct formalised in Section 2.3 is implemented in ScaFi [9,11], and describe, through examples, how aggregate processes based on `spawn` can be programmed in practice.

Background: ScaFi—Field Calculus in Scala. ScaFi (Scala Fields) is a development toolkit for aggregate systems in the Scala programming language. It provides a Scala-internal domain-specific language (DSL) – i.e., an API masked as an "embedded language"– and library of functions for programming with fields, as well as other development tools (e.g., for simulation). In ScaFi, the field constructs introduced in Section 2.1 are captured by the following interface:

```
trait Constructs {
  def rep[A](init: => A)(fun: A => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
  def branch[A](cond: => Boolean)(th: => A)(el: => A)
  def mid: ID
  ...
}
```

Method `branch` stands for field construct `if` (as the latter is a reserved keyword in Scala), `nbr` expressions are to be used within the `expr` passed to `foldhood` (used to aggregate over neighbours), and `mid` is a sensor giving the local device identifier. By ScaFi expressions one essentially defines "scripts" that specify whole fields at the macro-level: then, such scripts will be properly executed by each node/actor [11], following FC's operational semantics. A full introduction of ScaFi is outside the scope of this paper: it is deeply covered, e.g., in [9].

3.1 Aggregate Processes in ScaFi

The `spawn` primitive supports our notion of aggregate processes by handling activation, propagation, merging, and disposal of process instances (for a specified kind of process). Coherently with the formalisation in Section 2, it has signature:

```
def spawn[K,A,R](process: K => A => (R,Boolean),
                 newKeys: Set[K],
                 args: A): Map[K,R]
```

It is a generic function, parametrised by three types: *(i)* `K`, the type of process *keys*; *(ii)* `A`, the type of process *arguments* (or inputs); *(iii)* `R`, the type of process *result*. The function accepts three formal parameters and works as formalised in previous section. Note that a process key has a twofold role: it works both as a *process identifier* (PID) and as *initialisation* or *construction parameter*. When different construction parameters should result in different process instances, it is sufficient to instantiate type `K` with a data structure type including both pieces of information and with proper equality semantics. Function `spawn` accepts a *set* of keys to allow *generation* of zero or more process instances in the current round. Notice that if a new key already belongs to the set of active processes, there will be no actual generation (or restart) but *merging* instead, since identity is the same as an existing process instance. Finally, note also that the outcome of `spawn` (a map from process keys to process result values) can in turn be used to fork other process instances or as input for other processes; i.e., the basic means for processes to interact is to connect the corresponding `spawn`s with data.

In the following, we discuss programming and management of aggregate processes activated through `spawn`.

3.2 Process Generation, Expansion/Shrinking, and Termination

Generating process instances is just a matter of creating a field of keysets that become non-empty as soon as the proper space-time event has been recognised (e.g., spatial conditions on sensors data and computation, or timers firing) [34]. Then, by `spawn`, every process instance is *automatically propagated by all the participating devices to their neighbours*. However, it is possible to regulate the shape of such "computational bubble" by dictating conditions by which a device must return status `false` (i.e., meaning *external* to the bubble)—as mentioned, this indicates the willingness to *stop* computing (i.e., participate in) the process. That is, only devices that return status `true` (i.e., *internal*) will propagate the process. Moreover, such a propagation happens continuously: so, a device that exited from a process may execute it again in the future. In particular, the *border* (or *fence*) of a process bubble is given by the set of all the devices that are external but have at least one neighbour which is internal. As long as a node is in the fence, it continuously re-acquires and immediately quits from the process instance: this repeated evaluation of the border is what ultimately enables a spatial extension of the process bubble (*expansion*). Conversely, a process bubble gets restricted (*shrinking*) when internal nodes become external.

A process instance *terminates* when all the devices quit by returning status `false`. Implementing process termination may not be trivial, since proper (local or global) conditions must be defined so that the "collapsing force" can overtake the "propagation force"; i.e., precautions should be taken so that external devices do not re-acquire the process: the border should steadily shrink, also considering temporary network partitions and transient recoverable failures from devices.

Example: time replication. In [29], a technique based on time replication for improving the dynamics of gossip is presented. It works by keeping k running replicates of a gossip computation executing concurrently, each alive for a certain

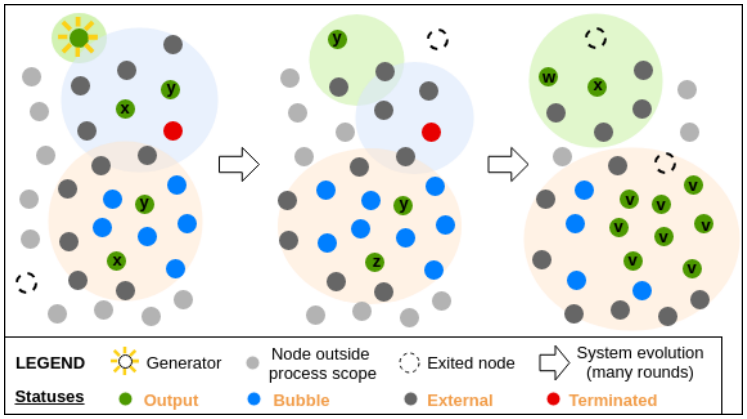

Fig. 2: Graphical example of the evolution of a system of processes and the role of statuses in `statusSpawn`. The green bubble springs into existence; the blue bubble dissolves after termination is initiated by a node; the orange bubble expands. Only output nodes will yield a value. Bubbles may of course overlap (i.e., a node may participate, with different statuses, to multiple processes) and the dynamics can be arbitrarily complex (because of mobility, failures, and local decisions).

amount of time. New instances are activated with interval p, staggered in time. The whole computation always returns the result of the oldest active replicate. This is intended to improve the dynamics of algorithms, providing an intrinsic refresh mechanism that smoothly propagates to the output. With `spawn`, it is trivial to design a `replicated` function that provides process replication in time.

```
def replicated[A,R](proc: A => R)(argument: A, p: Double, k: Int) = {
  val lastPid = clock(p, dt())
  spawn[Long,A,R](pid => arg => (proc(arg), pid > lastPid+k),
                  Set(lastPid), argument)
}
```

`clock` is a distributed time-aware counter [29] (whose synchronicity depends on the implementation) yielding an increasing number i at each interval p that represents the PID of the i-th replica. Notably, in this case, every device can locally determine when it must quit a process instance; moreover, the exit condition based on PID numbering (`pid > lastPid+k`) prevents process reentrance. Section 4.2 provides an empirical evaluation of the behaviour of function `replicated`.

3.3 More Expressive Process Definitions

Managing processes upon `spawn` revolves around specifying the logic for input/output, creation, evolution, and termination of processes instances. One approach to make such code more declarative consists of programming process behaviour so as to specify additional information w.r.t. just a boolean status/flag: more expressive `Status`es can be mapped to the flag and can be used to activate advanced behaviours. To do so, a higher-level function `statusSpawn` can be

considered, based on a `Status` value that indicates the "stance" of the current device w.r.t. the process instances at hand (see Figure 2): `Output` corresponds to flag `true` in `spawn`; `External` corresponds to flag `false` in `spawn`; `Bubble` means the device participates to the process but is not interested in the output (i.e., the process entry can be discarded); and `Terminated` means the device is willing to close the process instance (i.e., it triggers a shutdown behaviour).

Example: multi-gradient. The problem described in Section 2.2 of activating a spatially-limited gradient computation for each device where sensor `isSrc` gives true, and deactivating it when it stops doing so, can be solved as follows:

```
statusSpawn[ID,Double,Double](src => limit =>
  gradient(src==mid,nbrRange) match { // consider the usual gradient
    case g if src==mid && !isSrc => (g, Terminated) // close if src quits
    case g if g>limit => (g, External)              // out of bubble
    case g => (g, Output)                           // in bubble
  },
  newKeys = if(isSrc) Set(mid) else Set.empty,
  args = maxExtension)
```

4 Case Studies

In this section, we exercise the constructs previously introduced by presenting two application examples. One goal is to demonstrate the soundness of our implementation. Moreover, our empirical evaluation will also show that, orderly: *(i)* in certain cases, aggregate processes can greatly limit the consumption of computational resources while retaining a reasonable quality of service (QoS); *(ii)* in certain cases, powerful meta-algorithms enabled by aggregate processes can improve the dynamics of distributed computations. We implemented both scenarios with the Alchemist simulator [30], which already provides ScaFi support [9]; the results are the average over 101 runs. For the sake of reproducibility, the source code and instructions for running experiments are publicly available (`https://bitbucket.org/metaphori/experiment-spawn`).

4.1 Opportunistic Instant Messaging

Motivation. The possibility of communicating by delivering messages regardless the presence of a conventional Internet access has recently gained attention as a mean to work around censorship (`http://archive.is/C3niO`) as well as in situations with limited access to the global network—e.g., in rural areas, or during urban events when the network capability is overtaken. We here consider a simple messaging application where a source device (aka *sender*) wants to deliver a payload to a peer device (aka *recipient*, *target*, or *destination*) in a hop-by-hop fashion by exploiting nearby devices as relays. The source device only knows the identifier of its recipient: it is not aware of its physical location, nor of viable routes. Our goal is to show how aggregate processes can support this kind

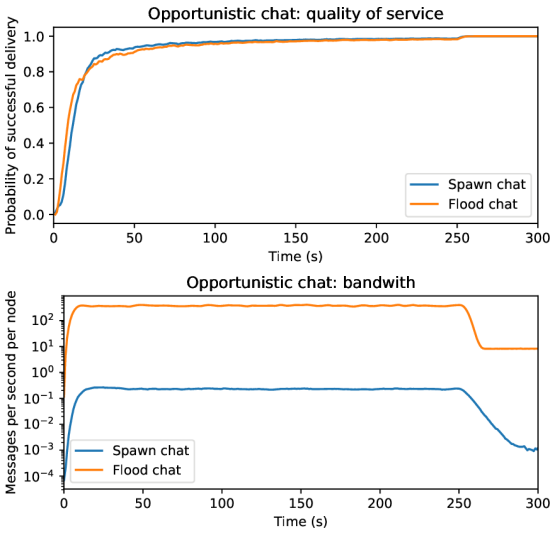

Fig. 3: Evaluation of the opportunistic chat algorithms. The figure on top shows similar performance for the two algorithms, with the *flood chat* featuring a slightly better delivery time for the payloads (as it intercepts the optimal path among others). However, as the bottom figure depicts, *spawn chat* requires orders of magnitude less resources due to the algorithm executing on a bounded area (i.e., by involving only a subset of system devices for each message delivery process).

of application (with multiple concurrent messages) while limiting the number of devices involved in message delivery, leading to bandwidth savings (and energy savings in turn).

Setup. We compare two aggregate implementations of such messaging system. The first implementation, called *flood chat*, simply broadcasts the payload to all neighbours. In spite of an in-place garbage collection system, however, this strategy may end up dispatching the message towards directions far-off the optimal path, burdening the network. The second implementation, *spawn chat*, leverages `spawn` in order to reduce the impact on the network infrastructure by electing a node as coordinator, then creating an aggregate process connecting the source and the coordinator and the coordinator and the destination, and finally delivering the message along such support. In this experiment, we naively choose a coordinator randomly, but better strategies could be deployed to improve over this configuration. The experiment is simulated on a mesh network of one thousand devices randomly deployed in the urban area of Cesena, in Italy. We simulate the creation and delivery of messages among randomly chosen nodes, with one message per second generated on average by the whole network in time window $[0, 250]$; devices execute rounds asynchronously at an average of 1Hz. We gather a measure of QoS and a measure of resource usage. We use the probability of delivering a message with time as a QoS measure, and we measure the number of payloads sent by each node as a measure of impact on performance. In doing so, we suppose payload makes up for the largest part of the communication (as is typically the case when multimedia data are exchanged).

Results. Figure 3 shows experimental results. The two implementations achieve a very similar QoS, with the flood implementation being faster on average. This is expected, as flooding the whole network also implies sending through the fastest path. The difference, however, is relatively small and, on the contrary, we

```
def gossipNaive[T](value: T)(implicit ev: Bounded[T]) = rep(value)(max =>
    ev.max(value, maxHoodPlus(nbr(ev.max(max, value)))))

def gossipReplicated[T:Bounded](value: T, p: Double, k: Int) =
  (replicated{ gossipNaive[T] }(value,p,k) // returns a Map[Long,Double]
   + (Long.MaxValue -> value) // default, lowest-priority entry of the map
  ).minBy[Long](_._1)._2 // projects the value of instance with min PID
```

Fig. 4: Code of the gossip algorithms used in the reconnaissance case study.

see the *spawn chat* affords a dramatic decrease in bandwidth usage (by properly constraining the expansion of message delivery bubbles), despite the simplistic selection of the coordinating device.

4.2 Reconnaissance with a Drone Swarm

Motivation. Performing reconnaissance of areas with hindrances to access and movement such as forests, steep climbs, or dangerous conditions (e.g. extreme weather and fire) can be a very difficult task for ground-based teams. In those cases, swarms of unmanned airborne vehicles (UAVs) could be deployed to quickly gather information [6]. One scenario in which such systems are particularly interesting is fire monitoring [12]. With this case study, we show how aggregate processes enable easy programming of a form of gossip that supports a precise collective estimation of risk in dynamic scenarios.

Setup. We simulate a swarm of 200 UAVs in charge of monitoring the area of Mount Vesuvius in Italy, which has been heavily hit by wildfires in 2017 (`http://archive.is/j3lsm`). UAVs follow a simple exploration strategy: they all start from the same base station on the southern side of the volcano, they visit a randomly generated sequence of ten waypoints, and once done they come back to the station for refuelling and maintenance. UAVs sense their surroundings once per second and assess the local situation by measuring the risk of fire. The

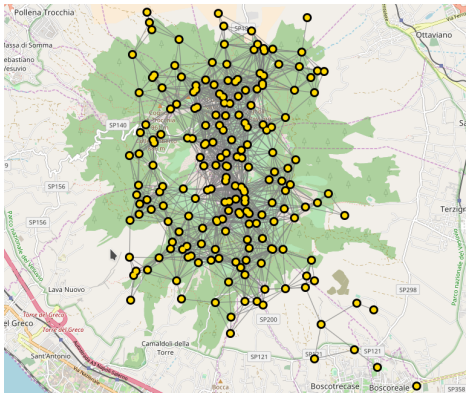

Fig. 5: Snapshot of the UAV swarm surveying the Vesuvius area as simulated in Alchemist. Yellow dots are UAVs. Grey lines depict direct drone-to-drone communication. Drones travel at an average speed of $130\frac{km}{h}$, in line with the cruise speed performance of existing military-grade UAVs (see `http://archive.is/8zar5`), and communicate with other drones within 1km distance in line-of-sight. Forming a dynamic mesh network using UAV-to-UAV communication is feasible [19], although challenging [22].

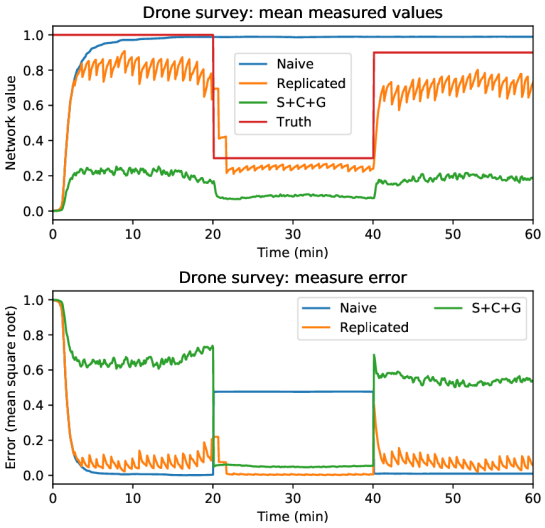

Fig. 6: Evaluation of the gossip algorithms in the UAV reconnaissance scenario. The figure on top shows expected values and measures performed by the competing algorithms. The bottom figure measures the error as root mean square: $\sqrt{\frac{\sum_n (v_n - a)^2}{n}}$ where n device count, a actual value, and v_n value at the n-th device. The naive gossip cannot cope with danger reduction, S+C+G cannot cope with the volatility of the network, while replicated gossip provides a good estimate while being to cope with changes.

goal of the swarm is to agree on the area with the highest risk of fire and report the information back to the station for deployment of ground intervention. A snapshot of the drones performing the reconnaissance is provided in Figure 5. In this paper, we are not concerned with realistic modelling of fire dynamics: we designed the risk of fire to be maximum in a random point of the surveyed area for 20 minutes; the risk then drops (e.g. due to a successful fire-fighting operation), with the new maximum (lower than the previous) being in another randomly generated coordinate; after further 20 minutes the risk sharply increases again to on a third random coordinate. We compare three approaches: *(i) naive gossip*, a simple implementation of a gossip protocol; *(ii) S+C+G*, a more elaborated algorithm – based on self-stabilising building blocks [34] – that elects a leader, aggregates the information towards it, then spreads it to the whole network by broadcast; *(iii) replicated gossip*, which replicates the first algorithm over time (as per [29]) and whose implementation, shown in Figure 4, uses function `replicated` (defined in Section 3 upon `spawn`).

Results. Results are shown in Figure 6. The naive gossip algorithm quickly converges to the correct value, but then fails at detecting the conclusion of the dangerous situation: it is bound to the highest peak detected, and so it is unsuitable for evolving scenarios. S+C+G can adapt to changes, but it is very sensitive to changes in the network structure: data gets aggregated along a spanning tree generated from the dynamically chosen coordinator, but in a network of fast-moving airborne drones such structure gets continuously disrupted. Here the `spawn`-based replicated gossip performs best, as it conjugates the stability of the naive gossip algorithm with the ability to cope with reductions in the sensed values. The algorithm in this case provides underestimates, as it reports the highest peak sensed in the time span of validity of a replicate, and drones rarely explore the exact spot where the problem is located, but rather get in its proximity.

5 Conclusions, Related and Future Work

In this paper, we have proposed and implemented a notion of aggregate processes to model dynamic, concurrent collective adaptive behaviours carried out by dynamic formations of devices—hence extending over field calculus and ScaFi.

Various spacetime- and macro-programming models have been developed across a wide variety of applications, which can potentially support mechanisms of aggregate processes. The survey [35] describes the historical evolution of "aggregate computing" from research in distributed systems, coordination languages, and spatial computing. In particular, four main clusters of approaches can be identified: *(i)* "bottom-up" approaches, such as TOTA [26], and Hood [37], that abstract individual networked devices; *(ii)* languages for expressing spatial and geometric patterns, such as GPL [14] and OSL [27]; *(iii)* languages for streaming and summarising information over space-time regions, such as Regiment [28] and TinyLime [15] and *(iv)* general purpose space-time computing models, such as MGS [20], the field calculus [4], and the Soft Mu-calculus for Computational fields (SMuC) [25]. Other works, often more generic and less operational, include models and languages for programming ensembles, such as SCEL [17], and process algebras (cf., the SAPERE approach [39]).

Multi-agent systems can bring agents together according to multiple organisational paradigms [23]. With aggregate processes, it is possible to program the logic of group formation so as to implement various grouping strategies. In the messaging case study, e.g., a dynamic, goal-directed *team* of devices is formed just to to connect senders with recipients, dissolving when the task is completed.

Related to the specifics of process execution, there are different models which aims at simplifying programming of multiple computing nodes as well as analysis of resulting programs. For instance, in the Bulk Synchronous Parallel (BSP) model [33], computations are structured as sequences of rounds followed by synchronisation steps; large-scale graph processing frameworks such as Apache Giraph [13] are inspired by BSP. Modern distributed data processing models (e.g., MapReduce [18] and derived ones) also abstract away network structure and trade performance for constrained programming schemas. By another perspective, works on service computing [7] tailored to dynamic ad-hoc environments [21] are also relevant but usually neglect the collective dimension and rarely consider open-ended situated activities. The service perspective connects also to utility computing and related efforts for abstracting and automatically managing networking and hardware infrastructure [32]—aggregate processes, by admitting diverse computation partitioning schemas [36], foster this vision.

In future work, we would like to use processes for advanced distributed coordination scenarios and implement a support for dynamic relocation of aggregate processes across a full IoT/Edge/Fog/Cloud stack. Further experimentation will be key to fully develop a theory of aggregate processes (e.g. in the style of π-calculus and its derivatives) as well as fully-fledged API and platform support.

Acknowledgements We thank the anonymous COORDINATION referees for their comments and suggestions on improving the presentation of the paper.

References

1. Audrito, G., Beal, J., Damiani, F., Viroli, M.: Space-time universality of field calculus. In: 20th International Conference on Coordination Models and Languages. LNCS, vol. 10852, pp. 1–20. Springer (2018)
2. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2017). pp. 91–100. IEEE (2017)
3. Audrito, G., Damiani, F., Viroli, M., Casadei, R.: Run-time management of computation domains in field calculus. In: 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 192–197. IEEE (2016)
4. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic **20**(1), 5:1–5:55 (2019)
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer **48**(9), 22–30 (2015)
6. Beal, J., Usbeck, K., Loyall, J., Rowe, M., Metzler, J.: Adaptive opportunistic airborne sensor sharing. ACM Transactions on Autonomous and Adaptive Systems **13**(1) (2018)
7. Bouguettaya, A., Singh, M., Huhns, M., Sheng, Q.Z., et al.: A service computing manifesto: the next 10 years. Communications of the ACM **60**(4), 64–72 (2017)
8. Casadei, R., Fortino, G., Pianini, D., Russo, W., Savaglio, C., Viroli, M.: Modelling and Simulation of Opportunistic IoT Services with Aggregate Computing. Future Generation Computer Systems **91**, 252—262 (2018)
9. Casadei, R., Pianini, D., Viroli, M.: Simulating large-scale aggregate MASs with Alchemist and Scala. In: FedCSIS Proceedings. pp. 1495–1504. IEEE (2016)
10. Casadei, R., Viroli, M.: Collective abstractions and platforms for large-scale self-adaptive IoT. In: 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 106–111. IEEE (2018)
11. Casadei, R., Viroli, M.: Programming actor-based collective adaptive systems. In: Programming with Actors, LNCS, vol. 10789, pp. 94–122. Springer (2018)
12. Casbeer, D.W., Kingston, D.B., Beard, R.W., McLain, T.W.: Cooperative forest fire surveillance using a team of small unmanned air vehicles. International Journal of Systems Science **37**(6), 351–360 (2006)
13. Ching, A., Edunov, S., Kabiljo, M., et al.: One trillion edges: Graph processing at facebook-scale. VLDB Endowment, Proceedings **8**(12), 1804–1815 (2015)
14. Coore, D.: Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer. Ph.D. thesis, MIT (1999)
15. Curino, C., Giani, M., Giorgetta, M., Giusti, A., et al.: Mobile data collection in sensor networks: The TinyLime middleware. Pervasive and Mobile Computing **4**, 446–469 (2005)
16. Damiani, F., Viroli, M.: Type-based self-stabilisation for computational fields. Logical Methods in Computer Science **11**(4) (2015)
17. De Nicola, R., et al.: The SCEL language: Design, implementation, verification. In: Software Engineering for Collective Autonomic Systems: The ASCENS Approach, pp. 3–71. Springer (2015)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
19. Frew, E., Brown, T.: Airborne communication networks for small unmanned aircraft systems. Proceedings of the IEEE **96**(12), 2008–2027 (dec 2008)

20. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computations in space and space in computations. In: Unconventional Programming Paradigms, LNCS, vol. 3566, pp. 137–152. Springer, Berlin (2005)
21. Groba, C., Clarke, S.: Opportunistic service composition in dynamic ad hoc environments. IEEE Trans. on Services Computing **7**(4), 642–653 (2014)
22. Gupta, L., Jain, R., Vaszkun, G.: Survey of important issues in UAV communication networks. IEEE Communications Surveys & Tutorials **18**(2), 1123–1152 (2016)
23. Horling, B., Lesser, V.: A survey of multi-agent organizational paradigms. The Knowledge engineering review **19**(4), 281–316 (2004)
24. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3), 396–450 (2001)
25. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Logical Methods in Computer Science **13**(1) (2017)
26. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Transactions on Software Engineering Methodologies **18**(4), 1–56 (2009)
27. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. Ph.D. thesis, MIT, Cambridge, MA, USA (2001)
28. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: WS on Data Managem. for Sensor Nets. pp. 78–87 (2004)
29. Pianini, D., Beal, J., Viroli, M.: Improving gossip dynamics through overlapping replicates. In: 18th International Conference on Coordination Models and Languages. LNCS, vol. 9686, pp. 192–207. Springer (2016)
30. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. Journal of Simulation **7**(3), 202–215 (2013)
31. Shi, W., Dustdar, S.: The promise of edge computing. IEEE Computer **49**(5), 78–81 (2016)
32. Truong, H.L., Dustdar, S.: Principles for engineering IoT cloud systems. IEEE Cloud Computing **2**(2), 68–76 (2015)
33. Valiant, L.: A bridging model for parallel computation. Communications of ACM **33**(8), 103–111 (1990)
34. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modelling and Computer Simulation **28**(2) (2018)
35. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From field-based coordination to aggregate computing. In: 20th Int. Conf. on Coordination Models and Languages. LNCS, vol. 10852, pp. 252–279. Springer (2018)
36. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: ACM UbiComp: Adjunct. pp. 1321–1326. ACM (2016)
37. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Conf. on Mobile systems, applications, and services. ACM (2004)
38. Zambonelli, F.: Toward sociotechnical urban superorganisms. IEEE Computer **45**(8), 76–78 (2012)
39. Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., et al.: Developing pervasive multi-agent systems with nature-inspired coordination. Pervasive and Mobile Computing (2014)