IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

**From field-based coordination to aggregate computing**

**This is the author's manuscript**

*Original Citation:*

*Availability:*

This version is available http://hdl.handle.net/2318/1671341 since 2018-07-25T16:14:58Z

*Publisher:*

Springer Verlag

*Published version:*

DOI:10.1007/978-3-319-92408-3_12

*Terms of use:*

Open Access

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

(Article begins on next page)

25 April 2024

# From Field-Based Coordination to Aggregate Computing*

Mirko Viroli[1], Jacob Beal[2], Ferruccio Damiani[3], Giorgio Audrito[3], Roberto Casadei[1], and Danilo Pianini[1]

[1] Alma Mater Studiorum–Università di Bologna, Italy
{mirko.viroli,roby.casadei,danilo.pianini}@unibo.it
[2] Raytheon BBN Technologies, USA
jakebeal@ieee.org
[3] Università di Torino, Italy
damiani@di.unito.it,audrito@di.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 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 provably resilient program components. In this survey, we trace the development and antecedents of field calculus, review the current state of aggregate computing theory and practice, and discuss a roadmap of current research directions that we believe can significantly impact the agenda of coordination models and languages.

## 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

* This work has been partially supported by: EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; ICT COST Action IC1402 ARVI (www.cost-arvi.eu); Ateneo/CSP D16D15000360005 project RunVar (runvar-project.di.unito.it). This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations.

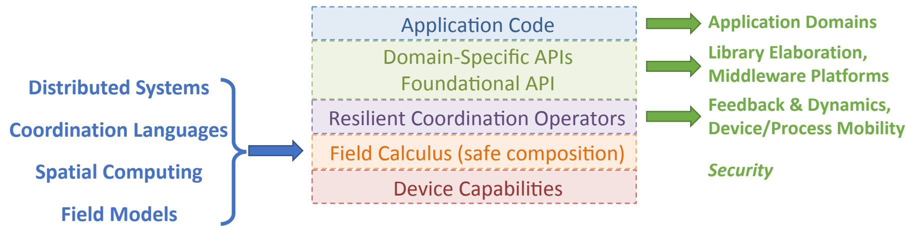


**Fig. 1.** This survey reviews the development of field calculus from its antecedents (left), the current state of aggregate computing theory and practice as layered abstractions based on field calculus (middle), and current research directions on top of field calculus with respect to challenges in coordination models and languages (right).

and coordinate the operation of such systems is to program and operate in terms of *aggregates* of devices rather than attempting to micro-manage 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 advantages in resilience, efficiency, and privacy.

Aggregate computing is an emerging approach, developed significantly within the coordination models and languages research community, that embraces this environment, building from a foundation on 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 to engineering coordination of complex distributed systems has been constructed, 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, both in theory, engineering methods and tools, and applications.

In this survey, we present a discussion of the past, present, and future of aggregate computing (Figure 1). Section 2 begins by tracing the development of aggregate computing through its antecedents both in coordination research and in other areas, culminating in development of the field calculus. Section 3 then discusses the current state of aggregate computing theory and practice across its various abstraction layers. Finally, Section 4 presents a roadmap of current research directions on top of field calculus and with respect to challenges in coordination models and languages, and Section 5 summarises and concludes.

## 2 Coordination, Self-Organisation, and Fields

In this section we review and discuss the conceptual, but also technical and technological, path that brought traditional coordination models for parallel computing, step-by-step to address the complexity of self-organising, large-scale deployed systems (Section 2.1). Then, we describe the emergence of field-based coordination (Section 2.2), and how, through the interaction with research works falling under the umbrella of space-based computation models (Section 2.3), this path ended up in the field calculus as discussed in next section.

### 2.1 Coordination towards Self-organisation

**Generative communication** Coordination models are rooted into the idea that interaction among multiple, independent, and autonomous software systems (processes, components, 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 [36], enabling so-called *generative communication*. Linda [52] is universally 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 by writing and retrieving, with a suspensive semantics (the requestor is blocked until the query is satisfiable), data in form of 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 to synchronise 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 logic tuple-space models, where software agents coordinate through first-order tuples, and tuple spaces can be programmed as first-order logic theories. A prominent example of such approach is Shared Prolog [23], 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", i.e., in the form of an application logic that can manipulate data in the shared space and the way 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 [24]. Among them, we find Law-Governed Interaction [65], MARS [26] and ReSpecT [71].

**Distribution** All these approaches, however, do not explicitly focus on distributed systems, but on the coordination of centralised local components. As software components get spread across the system topology, so multiple tuple spaces 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 [51] and TSpaces [97]. Lime [67], Klaim [44], and LogOp [63], take the approach a step further, by allowing to express the dynamic environment topology in a distributed setting, thus paving the way towards application of coordination models to pervasive computing system scenarios.

**Self-organising coordination** As coordination abstractions of various sorts (tuple spaces, channels, coordination artifacts [93,72]) are available in the 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 (as the ability of intercepting relevant events, and react to them so as to guarantee certain levels of overall system resilience). This calls for an approach of *self-organising coordination* [89], 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 one.

Coordination models following this approach typically take their inspiration from complex natural systems (from physics through chemistry all the way to ethology) and reuse their foundational mechanisms. A primary source of inspiration for these systems is to be found in biology (social animals, and insects in particular), whose foraging techniques inspire the mechanisms that regulate coordination [27,80,78]. For instance, SwarmLinda [80] is a tuple-based middleware that brings the collective intelligence displayed by swarms of ants to computational mechanisms to guarantee efficient retrieval. Tuples are handled as sort of pheromones or items that ants (agents) continuously and opportunistically relocate. Chemical-inspiration is used in [87,88] to regulate the "activity level" of tuples, which drives the likelihood of their retrieval as well as their propagation rate. Ecology-inspiration is instead used in [81] to inject competition, composition, and disposal behaviour in the context of coordination of pervasive computing services.

### 2.2 Field-based coordination

Another important natural source of inspiration comes from physics: physics-inspired self-organising coordination systems rely on the notion of "field" (grav-

itational field, electromagnetic field), which essentially provides a framework to handle (create, manipulate, combine) global-level, distributed data structures.

A notion of *coordination field* (or co-field) was initially proposed in [62] 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 [61] 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 [79], is an extension to traditional Linda tuple spaces with the goal of supporting resource discovery in a pervasive system, relying on ideas inspired by 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 (i.e., process algebras and transition systems) is the $\sigma\tau$-Linda model [94], where agents can inject into the space "processes" that spread, collect and decay tuples, ultimately sustaining fields of tuples.

### 2.3 Spatial computing approaches: towards the field calculus

More or less independently from 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 [14,46,64,73] 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., SCEL [45], Hood [96], Butera's "paintable computing" [25], and Meld [1]), spatial patterns and languages (e.g., Growing Point Language [35], geometric patterns in Origami Shape Language [68], self-healing geometries [34], or universal patterns [98]), tools to summarise and stream information over regions of space and time (e.g., TinyDB [60], Cougar [99], TinyLime [37], and Regiment [69]), 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 [57], systolic computing [48]) and topological computing (e.g., MGS [53,54]). Among them, space-time

computing models based on the notion of computational fields were initially proposed in [12] and implemented in the Proto language. 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.

## 3 From Field Calculus to Aggregate Computing

In this section, we discuss the current state of the art in aggregate computing, with the goal of presenting the full spectrum of results achieved without going into deep technical details—the reader can access code examples and tutorials, as well as formalisation of semantics, from the references provided. We begin with a review of its mathematical core in field calculus (Section 3.1), then discuss the construction of implementations of field calculus as the domain specific language Protelis (Section 3.2) and Scala support ScaFi (Section 3.3). Finally, we discuss the layered abstractions of aggregate programming built upon these foundations, from resilient operators to pragmatic libraries (Section 3.4).

### 3.1 Field Calculus

**Basic calculus** The field calculus (FC) has been proposed in [92] as a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields:[4] functional composition of fields, functions over 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 aggregate system behaviour of a network of devices (where a dynamic neighbouring relation represents physical or logical proximity) by 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 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 calculus expression `e` specifies a mapping (i.e., the computational field) associating each computation round of each device to the value that `e` assumes on 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 [41,92].

The distinguished interaction model of this approach has been first formalised in [92] by means of a small-step operational semantics modelling single device

[4] Much as $\lambda$-calculus [32] captures the essence of functional computation and FJ [55] the essence of class-based object-oriented programming.

```
// 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
distanceWithObs(deviceId == 0, senseObs())
```

**Fig. 2.** Example field calculus code

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. Field construction, propagation, and restriction are then supported by local evaluation "against" the evaluation trees received from neighbours. Accessing these values is allowed by two specialised constructs:

- `rep`($e_0$){(`x`)=>`e`} which retrieves the value `v` computed for the whole `rep` expression in the last evaluation round (the value produced by evaluating the expression $e_0$ is used at the first evaluation round) and updates it by the value produced by evaluating the expression obtained from `e` by replacing the occurrences of `x` by `v`;
- `nbr`{`e`}, which gathers the values computed by neighbours for expression `e` (from the respective evaluation trees) in their last round of computation into a *neighbouring field value*, which is a map from neighbour device identifiers to their correspondent values.

These constructs are backened by a data gathering mechanism accomplished through a process called *alignment*, which ensures appropriate message matching, i.e., that no two different `nbr` expressions can inadvertently "swap" their respective messages. This has the notable consequence that the two branches of an `if` statement in field calculus are executed in isolation: devices computing the "then" branch cannot communicate with a device computing the "else" branch, and viceversa.

Consider as an example Figure 2. Function `distanceTo` takes as argument a field of booleans `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, computing relaxation of triangle inequality; namely: repetitively, and starting from infinity (construct `rep`) everywhere, the distance on any node gets updated to 0 on source nodes (function `mux(c,t,e)` is a purely functional multiplexer which chooses `t` if `c` is `true`, or `e` otherwise), and elsewhere to the minimum

(built-in `minHood`) of neighbours' 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 reuses computation of `distanceTo`. Because of alignment, the set of considered neighbours 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 `id` equal to `0`, circumventing the devices where `senseObs` gives true.

The work in [41] (which is an extended and revised version of [92]) presents a type system, used to intercept ill-formed field-calculus programs. The type system, which builds on the HindleyMilner type system [39] for ML-like functional languages, is specified by a set of syntax-directed type inference rules. Being syntax-directed, the rules straightforwardly specify a variant of the Hindley-Milner type inference algorithm [39]. Namely, an algorithm that given a field calculus expression and type assumptions for its free variables: either fails (if the expression cannon 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* and *types for functions* (built-in operators and user-defined functions)—this reflects the fact that the 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* (e.g., the values produced by numerical literals) and types for *neighbouring field values* (e.g., the values produced by `nbr`-expressions).

The type system is proved to guarantee the following two properties:

- *Domain alignment:* On each device, the domain of every neighbouring field 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 terminates.

**Higher-order field calculus** The higher-order field calculus (HFC) [42] (see also [84]) 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 parameterised 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 change over time (via `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 the 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 insight of HFC, enabling first-class functions to be implemented with relatively minimal complexity.

In [42] 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 subexpressions. Moreover, [42] 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.

**Behavioural properties** Since HFC is designed as a general-purpose language for spatially distributed computations, its semantics and type system guarantees do not prevent the formulation of ill-behaving programs. 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 [47,59,58] plays a central role. This notion, defined in terms of properties of the transition system of network evolution, 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 [40] (which is an extended version of [91]), a first self-stabilising fragment is isolated through a *spreading* operator, which minimises neighbour val-

ues 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. However, more self-stabilising programs and existing "building block" implementations are covered by the larger self-stabilising fragment introduced in [83] (which is an extended version of [86]). This fragment restricts the usage of `rep` statements to three specific patterns (converging, acyclic and minimising `rep`), roughly corresponding to the three main building blocks (time evolution, aggregation, distance estimation). 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 a same program: operational semantics (local viewpoint), denotational semantics (global viewpoint), and eventual behaviour (limit viewpoint).

A fourth "continuous" viewpoint is considered in [20]: 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 part, *provided* that the inputs are constant (except for a transitory initial part). Eventual consistency can then be proved for all programs expressible in the GPI (gradient-following path integral) calculus, that 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.

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 [59], or model-based analysis such as [7].

### 3.2 Protelis: a DSL for field calculus

The concrete usage of HFC in application development is conditioned by the availability of practical languages, embedding an interpreter or compiler, as well as handling runtime aspects such as communication, interfacing with the operating system, and integration with existing software. Protelis [77] provides one such implementation, including: (i) a concrete HFC 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 the HFC semantics. Then, the program, along with an execution context, is fed to the 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 capabilities and the communication system. This architecture has been proven to make the language easy to port across diverse contexts, both simulated (Alchemist [76] and NASA World Wind [21]) and real-world [33].

The entire Protelis infrastructure is developed in Java and hosted on the Java Virtual Machine (JVM). The motivation behind one such 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 of writing new libraries for common tasks.

The model-to-model translation between the Protelis syntax and the HFC interpreter is operated by leveraging the Xtext framework [22]. Along the parser machinery, this framework is 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 idea 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 3 offers a panorama on the ordinary and field-calculus specific features of Protelis, including the ability of 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 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`.

```
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 visibile
  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
}
```

**Fig. 3.** Example Protelis code showcasing detailed syntactic aspects

A relevant asset of Protelis is its recently developed library "protelis-lang" [50], streamlining the implementation of several algorithms found in literature devoted to development of distributed systems. Among others, it includes several implementations of self-stabilising building functions [18,83], such as `distanceTo` to estimate distances, `broadcast` to send alerts, `summarize` to perform distributed sensing, and so on. Notably, the library also includes machinery for "aligning" 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 to new possibilities such as `multiInstance` [50], or allow for increased resilience and adaptation as in the case of `timeReplicated` [75].

### 3.3 ScaFi: an API for the Scala programming ecosystem

From a pragmatical viewpoint, it is highly desirable to bridge the gap between field calculus-based DSLs and mainstream programming platforms and

languages that embody, among others, the functional, object-oriented, and actor-based paradigms (i.e., the nowadays reference styles for in-the-small, in-the-large, and concurrent/distributed programming, respectively). Indeed, this can 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 [22], can require a lot of development and maintenance effort, since they must cover aspects ranging from language design to typing, and proper 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) [30], an aggregate computing framework including a field-calculus DSL internal to the Scala programming language [70], also integrated into the Alchemist meta-simulator [29], as well as an actor-based platform for distributed aggregate systems [31,90]. 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 to create fluent DSL-like APIs; and *(v)* prominent role in the scene of distributed computing frameworks (cf., Akka, Kafka, Spark). Concerning the platform perspective, instead, 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 [31].

Working with a general-purpose, multi-paradigm programming language like Scala gives to the hands of developers quite a lot of flexibility and power for what concerns design and implementation of field libraries and programs. Consider the example in Figure 4 for a taste of the programming style, including definition of a reusable block `G` (extending distance calculation [15,83]), type-class-style assumptions on arguments via context bound "`[V: Bounded]`", tuples by syntax `(.,.)`, and pattern matching (`case .. =>`).

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 instantiating 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.

```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)
}
```

**Fig. 4.** Example ScaFi code

### 3.4 Aggregate Programming

Building upon these theoretical and pragmatic foundations, aggregate programming [15] 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;
- mechanisms for robust coordination should be hidden "under the hood", where programmers are not required to interact with them.

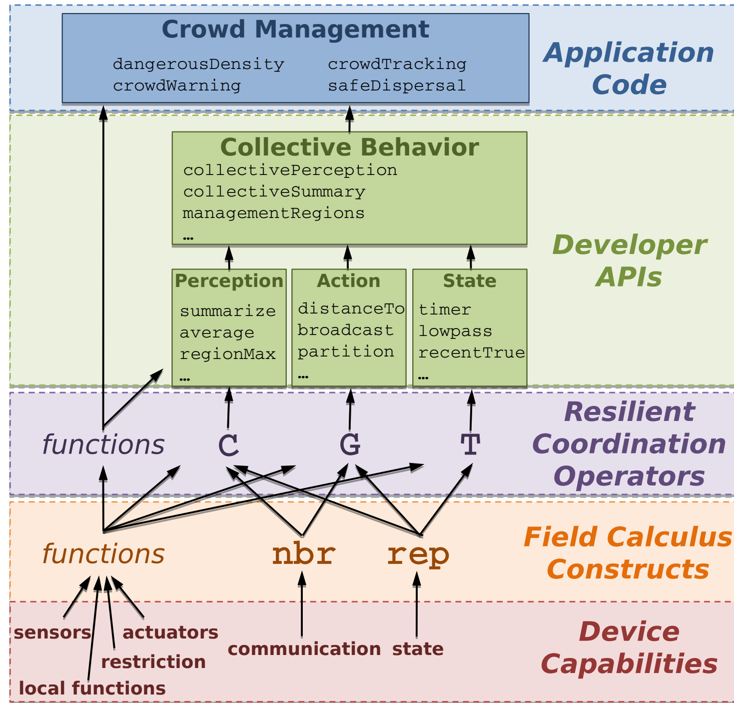


**Fig. 5.** Aggregate programming abstraction layers. The software and hardware capabilities of particular devices are used to implement aggregate-level field calculus constructs. These constructs are used to implement a limited set of building-block coordination operations with provable resilience properties, which are then wrapped and combined together to produce a user-friendly API for developing situated IoT (Internet-of-Things)systems—picture adapted from [15].

Field calculus (and 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.

Aggregate programming thus proposes two additional abstraction layers, as illustrated in Figure 5, 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 [18], it is inspired by the approach of combinatory logic [38], the catalogue of self-organisation primitives in [49], and work on self-stabilising fragments of the field calculus [40,83,91]. Notably, three key operators within this self-stabilising fragment cover a broad range of distributed coordination patterns: operator `G` is 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 [50,86] 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 [83,86].

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 interaction 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 to calculate 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 [13], 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 [8]. Several alternative gradient algorithms have been developed, addressing two main issues. Firstly, the recovery speed after an input discontinuity, which has first been bounded to $O(diameter)$ time by CRF (constraint and restoring force) gradient [13], further improved to optimal for algorithms with a single-path communication pattern by BIS (bounded information speed) gradient [5], and refined to optimality for algorithms with a multi-path communication pattern by SVD (stale values detection) gradient [3]. Secondly, the smoothness and resilience to noise in inputs, first addressed by FLEX (flexible) gradient [8] and then refined and combined with improved recovery speed by ULT (ultimate) gradient [3].

To empower the aggregate programming tool-chain, other building blocks have been proposed and refined besides from gradients: consensus algorithms [11], centrality measures [4], leader election and partitioning [18], and most notably, *collection*. The collection building block `C` progressively aggregates and summarises values spread throughout a network into a single value, e.g., the sum or other meaningful statistics. Based itself on distance estimation through gradients, a general *single-path* collection algorithm has been proposed in [18] granting self-stabilisation to a correct value, then *multi-path* collection has been

developed for improved resiliency in sum estimations [83], and finally refined to *weighted multi-path* collection [2] which is able to maintain acceptable whole network sums 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 [75] (with an appropriately tuned interval of replication), thus combining the resiliency of these algorithms with self-stabilisation requirements.

# 4 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 [85], 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 4.1), techniques to control dynamics (Section 4.2), management of mobile devices and processes (Section 4.3), development of software platforms (Section 4.4), security (Section 4.5), and applications (Section 4.6).

## 4.1 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. Some of these additions and refinements will be based on development of alternative implementations of core resilient building blocks (e.g., [2,5,75]), while others are expected to capturing common design patterns and necessary functionalities specific to particular application domains. No particular high-priority targets are suggested at present for this development, however. Instead, this process is expected to be a natural incremental progress 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.

## 4.2 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 [47,82] and eventual consistency [20]. 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. The mathematical frameworks for such tools are not straightforward to adapt for application to aggregate computing building blocks, but with careful work 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 [56] and providing stability analysis and tight convergence bounds for particular applications of the `G` operator [43] and `C` operator [66]. 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.

### 4.3 Mobility of Devices and Processes

Another key area for expansion of aggregate computing, both in theory and practice, is better handling of mobility, aboth 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. Instead, work to date has depended on the informal observation that "slow enough" mobility does not disrupt commonly used self-stabilising building blocks. 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., [6,9,10]). We may also consider systems in which the device is not the focus of mobility, but instead code and processes dynamically deploy, migrate, upgrade, and terminate during system operation, as considered for example in [15,95]. 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.

### 4.4 Software Platforms

Aggregate computing targets a number of application scenarios, generally characterised by inherent distribution, heterogeneity, mobility and lack of stable infrastructure (including computation, storage, and networking media). Hence, proper middleware or software platform is paramount to ease the development and deployment of applications as well as support their management at runtime [90]. Moreover, such a layer is the ideal place where to encapsulate cross-cutting concerns such as security, privacy, monitoring, fault tolerance and so on.

Though the problem of a middleware is common to almost any distributed computing effort, 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, consider the aggregate programming model: it achieves a certain degree of declarativity by abstracting 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 lot of flexibility on the platform side, which is free to apply optimisations of various sorts, from simpler (e.g., avoid broadcasting redundant messages) to more complex ones. In fact, the most relevant insight here is the ability of running aggregate computing systems according to different execution strategies [90], from fully peer-to-peer, where end-devices directly communicate between one another and run by themselves their piece of aggregate logic, to completely centralised solutions where, instead, end-devices act only as managers for sensors and actuators, sending perceptions upstream to one or more servers which run computations on their behalf and ultimately propagate actuation data downstream.

Crucially, this flexibility paves the way towards an opportunistic and QoS-driven exploitation of available infrastructural resources, as well as to intrinsic adaptation of application execution to forthcoming multi-layer architectures involving edge, fog, and cloud interfaces [90]—as required to deal with emerging IoT scenarios. For instance, an aggregate system specification can be mapped to a system of actors [31] 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 [90]. A lot of interesting future work is expected to be carried out in order to put such theory of adaptive execution coordination into practice.

### 4.5 Security

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 issues such as, for instance, the cost of security-related computational tasks in resource-constrained devices.

A number of security issues, not strictly related to coordination, 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 [42], which may be required in scenarios characterised by significant dynamicity requirements or demands for automatic deployment of new functionality. Another key theme is confidentiality: privacy properties on the propagated and collected data need to be understood and guaranteed, otherwise participation may be hindered. 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).

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 [28] 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. Much work remains, however, to develop these initial proposals into a fully articulated theory and practice for the security of aggregate computing systems that takes into account confidentiality, integrity, availability, and authenticity issues.

### 4.6 Applications and Pragmatics

Finally, the core goal all along for the aggregate computing research thrust has 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.

One key application area, previously discussed in [15] 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. 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 in this area 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.

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 [6] and [9]. 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.

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 [95], 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 [74] and agent-based sharing of airborne sensors [16,17]. 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.

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 [33], coordinating a checkpoint-based "rewind and replay" across

interacting services to undo the effects of a cyber-attack [19], and integrating applications across intermittently connected distributed cloud nodes [19]. 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 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.

In addition to the four 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.

## 5 Conclusions

Aggregate computing is a potentially powerful approach to the engineered 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.

## References


1. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: International Conference on Intelligent Robots and Systems (IROS). pp. 2794–2800. IEEE (2007)
2. Audrito, G., Bergamini, S.: Resilient blocks for summarising distributed data. In: ALP4IoT workshop, to appear on EPTCS online (2017)

3. 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). pp. 91–100. IEEE (2017)
4. Audrito, G., Damiani, F., Viroli, M.: Aggregate graph statistics. In: ALP4IoT workshop, to appear on EPTCS online (2017)
5. Audrito, G., Damiani, F., Viroli, M.: Optimally-self-healing distributed gradient structures through bounded information speed. In: International Conference on Coordination Models and Languages (COORDINATION). pp. 59–77. Springer (2017)
6. Bachrach, J., Beal, J., McLurkin, J.: Composable continuous-space programs for robotic swarms. Neural Computing and Applications 19(6), 825–847 (2010)
7. Bakhshi, R., Cloth, L., Fokkink, W., Haverkort, B.R.: Mean-field framework for performance evaluation of pushpull gossip protocols. Performance Evaluation 68(2), 157 – 179 (2011), advances in Quantitative Evaluation of Systems
8. Beal, J.: Flexible self-healing gradients. In: Symposium on Applied Computing. pp. 1197–1201. ACM (2009)
9. Beal, J.: A tactical command approach to human control of vehicle swarms. In: AAAI Fall Symposium: Human Control of Bioinspired Swarms (2012)
10. Beal, J.: Superdiffusive dispersion and mixing of swarms. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 10(2), Article 10 (2015)
11. Beal, J.: Trading accuracy for speed in approximate consensus. The Knowledge Engineering Review 31(4), 325–342 (2016)
12. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intelligent Systems 21, 10–19 (2006)
13. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Symposium on Applied computing. pp. 1969–1975. ACM (2008)
14. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013), a longer version available at: http://arxiv.org/abs/1202.5509
15. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer 48(9), 22–30 (2015)
16. Beal, J., Usbeck, K., Loyall, J., Metzler, J.: Opportunistic sharing of airborne sensors. In: International Conference on Distributed Computing in Sensor Systems (DCOSS). pp. 25–32. IEEE (2016)
17. Beal, J., Usbeck, K., Loyall, J., Rowe, M., Metzler, J.: Adaptive task reallocation for airborne sensor sharing. In: International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 168–173. IEEE (2016)
18. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: 8th International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW). pp. 8–13 (2014)
19. Beal, J., Viroli, M.: Aggregate programming: From foundations to applications. In: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems, pp. 233–260. Springer (2016)
20. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the Internet of Things. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 12(3), 12 (2017)
21. Bell, D.G., Kuehnel, F., Maxwell, C., Kim, R., Kasraie, K., Gaskins, T., Hogan, P., Coughlan, J.: NASA world wind: Opensource GIS for mission operations. In: Aerospace Conference. IEEE (2007)
22. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2E. Packt Publishing (2016)

23. Brogi, A., Ciancarini, P.: The concurrent language, Shared Prolog. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 99–123 (1991)
24. Busi, N., Ciancarini, P., Gorrieri, R., Zavattaro, G.: Coordination models: A guided tour. In: Coordination of Internet Agents: Models, Technologies, and Applications, chap. 1, pp. 6–24. Springer (2001)
25. Butera, W.: Programming a Paintable Computer. Ph.D. thesis, MIT, Cambridge, USA (2002)
26. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. IEEE Internet Computing 4(4), 26–35 (2000)
27. Casadei, M., Viroli, M., Gardelli, L.: On the collective sort problem for distributed tuple spaces. Science of Computer Programming 74(9), 702–722 (2009)
28. Casadei, R., Aldini, A., Viroli, M.: Combining trust and aggregate computing. In: Software Engineering and Formal Methods. pp. 507–522. Springer, Cham (2018)
29. Casadei, R., Pianini, D., Viroli, M.: Simulating large-scale aggregate MASs with Alchemist and Scala. In: Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 1495–1504. IEEE (2016)
30. Casadei, R., Viroli, M.: Towards aggregate programming in Scala. In: 1st Workshop on Programming Models and Languages for Distributed Computing. p. 5. ACM (2016)
31. Casadei, R., Viroli, M.: Programming actor-based collective adaptive systems. In: Programming with Actors - State-of-the-Art and Research Perspectives. Lecture Notes in Computer Science, vol. 10789. Springer (2018), to appear
32. Church, A.: A set of postulates for the foundation of logic. Annals of Mathematics 33(2), 346–366 (1932)
33. Clark, S.S., Beal, J., Pal, P.: Distributed recovery for enterprise services. In: 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). pp. 111–120. IEEE (2015)
34. Clement, L., Nagpal, R.: Self-assembly and self-repairing topologies. In: Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open (2003)
35. Coore, D.: Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer. Ph.D. thesis, MIT, Cambridge, MA, USA (1999)
36. Corkill, D.: Blackboard systems. Journal of AI Expert 9(6), 40–47 (1991)
37. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: The TinyLime middleware. Elsevier Pervasive and Mobile Computing Journal 4, 446–469 (2005)
38. Curry, H., Feys, R.: Combinatory logi. North-Holland (1958)
39. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Symposium on Principles of Programming Languages (POPL). pp. 207–212. ACM (1982)
40. Damiani, F., Viroli, M.: Type-based self-stabilisation for computational fields. Logical Methods in Computer Science 11(4) (2015)
41. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. Science of Computer Programming 117, 17 – 44 (2016)
42. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: A higher-order calculus of computational fields. In: Formal Techniques for Distributed Objects, Components, and Systems, Lecture Notes in Computer Science, vol. 9039, pp. 113–128. Springer (2015)
43. Dasgupta, S., Beal, J.: A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In: 55th IEEE Conference on Decision and Control (CDC). pp. 7282–7287. IEEE (2016)

44. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agent interaction and mobility. IEEE Transaction on Software Engineering (TOSE) 24(5), 315–330 (1998)
45. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 9(2), 7:1–7:29 (2014)
46. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1(2), 223–259 (2006)
47. Dolev, S.: Self-Stabilization. MIT Press (2000)
48. Engstrom, B.R., Cappello, P.R.: The SDEF programming system. Journal of Parallel and Distributed Computing 7(2), 201 – 231 (1989)
49. Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Natural Computing 12(1), 43–67 (2013)
50. Francia, M., Pianini, D., Beal, J., Viroli, M.: Towards a foundational API for resilient distributed systems design. In: International Workshops on Foundations and Applications of Self* Systems (FAS*W). IEEE (2017)
51. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices. The Jini Technology Series, Addison-Wesley Longman (1999)
52. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) 7(1), 80–112 (1985)
53. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational models for integrative and developmental biology. Tech. Rep. 72-2002, Univerite d'Evry, LaMI (2002)
54. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computations in space and space in computations. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, vol. 3566, pp. 137–152. Springer, Berlin (2005)
55. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(3), 396–450 (2001)
56. Kumar, A., Beal, J., Dasgupta, S., Mudumbai, R.: Toward predicting distributed systems dynamics. In: International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW). pp. 68–73. IEEE (2015)
57. Lasser, C., Massar, J., Miney, J., Dayton, L.: Starlisp Reference Manual. Thinking Machines Corporation (1988)
58. Lluch-Lafuente, A., Loreti, M., Montanari, U.: A fixpoint-based calculus for graph-shaped computational fields. In: 17th International Conference on Coordination Models and Languages (COORDINATION). pp. 101–116 (2015)
59. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Logical Methods in Computer Science 13(1) (2017)
60. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: A Tiny AGgregation service for ad-hoc sensor networks. SIGOPS Operating System Review 36(SI), 131–146 (2002)
61. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Transactions on Software Engineering Methodologies (TOSEM) 18(4), 1–56 (2009)

62. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In: Engineering Societies in the Agents World III. pp. 68–81. Springer (2003)
63. Menezes, R., Snyder, J.: Coordination of distributed components using LogOp. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). vol. 1, pp. 109–114. CSREA Press (2003)
64. Menezes, R., Tolksdorf, R.: Adaptiveness in Linda-based coordination models. In: Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering, LNAI, vol. 2977, pp. 212–232. Springer (2004)
65. Minsky, N.H., Ungureanu, V.: Law-Governed interaction: A coordination and control mechanism for heterogeneous distributed systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 9(3), 273–305 (2000)
66. Mo, Y., Beal, J., Dasgupta, S.: Error in self-stabilizing spanning-tree estimation of collective state. In: International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 1–6. IEEE (2017)
67. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(3), 279–328 (2006)
68. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. Ph.D. thesis, MIT, Cambridge, MA, USA (2001)
69. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87 (2004)
70. Odersky, M., Rompf, T.: Unifying functional and object-oriented programming with Scala. Communications of ACM 57(4), 76–86 (2014)
71. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming 41(3), 277–294 (2001)
72. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 286–293. IEEE Computer Society (2004)
73. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review 26(1), 53–59 (2011)
74. Parunak, H.V.D., Brueckner, S.: Concurrent modeling of alternative worlds with polyagents. In: Multi-Agent-Based Simulation VII, pp. 128–141. Springer (2007)
75. Pianini, D., Beal, J., Viroli, M.: Improving gossip dynamics through overlapping replicates. In: 18th International Conference on Coordination Models and Languages (COORDINATION). pp. 192–207. Springer (2016)
76. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation 7(3), 202–215 (2013)
77. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Symposium on Applied Computing. pp. 1846–1853. ACM (2015)
78. Pianini, D., Virruso, S., Menezes, R., Omicini, A., Viroli, M.: Self organization in coordination systems using a WordNet-based ontology. In: 4th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE (2010)
79. Stovall, D., Julien, C.: 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. pp. 1–10. ESSPE, ACM, New York, NY, USA (2007)

80. Tolksdorf, R., Menezes, R.: Using swarm intelligence in Linda systems. In: Engineering Societies in the Agents World IV, Lecture Notes in Computer Science, vol. 3071, pp. 519–519. Springer (2004)
81. Viroli, M.: On competitive self-composition in pervasive services. Science of Computer Programming 78(5), 556 – 568 (2013)
82. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. arXiv preprint arXiv:1711.08297 (2017)
83. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. to appear on ACM Transactions on Modeling and Computer Simulation (TOMACS) (2018)
84. Viroli, M., Audrito, G., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. arXiv preprint arXiv:1610.08116 (2016)
85. Viroli, M., Beal, J.: 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)
86. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). pp. 81–90 (September 2015)
87. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: Lecture Notes in Computer Science, pp. 143–162. Springer (2009)
88. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 6(2), 14:1 – 14:24 (2011)
89. Viroli, M., Casadei, M., Omicini, A.: A framework for modelling and implementing self-organising coordination. In: ACM Symposium on Applied Computing (SAC). pp. 1353–1360 (2009)
90. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct. pp. 1321–1326. ACM (2016)
91. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: 16th International Conference on Coordination Models and Languages (COORDINATION), LNCS, vol. 8459, pp. 163–178. Springer (2014)
92. Viroli, M., Damiani, F., Beal, J.: A calculus of computational fields. In: Advances in Service-Oriented and Cloud Computing, Communications in Computer and Information Science, vol. 393, pp. 114–128. Springer (2013)
93. Viroli, M., Omicini, A., Ricci, A.: Engineering MAS environment with artifacts. In: Weyns, D., Parunak, H.V.D., Michel, F. (eds.) 2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005). AAMAS 2005, Utrecht, The Netherlands (26 Jul 2005)
94. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: An adaptive coordination model for mobile ad-hoc environments. In: 14th International Conference on Coordination Models and Languages (COORDINATION). pp. 212–229 (2012)
95. Viroli, M., Pianini, D., Ricci, A., Croatti, A.: Aggregate plans for multiagent systems. International Journal of Agent-Oriented Software Engineering 4(5), 336–365 (2017)
96. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: 2nd International Conference on Mobile systems, applications, and services. ACM (2004)
97. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T Spaces. IBM Journal of Research and Development 37(3 – Java Techonology), 454–474 (1998)

98. Yamins, D.: A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems. Ph.D. thesis, Harvard, Cambridge, MA, USA (2007)
99. Yao, Y., Gehrke, J.: The Cougar approach to in-network query processing in sensor networks. ACM Sigmod record 31(3), 9–18 (2002)