

# Towards Agent Aggregates: Perspectives and Challenges

Mirko Viroli<sup>(✉)</sup> and Alessandro Ricci

Alma Mater Studiorum – Università di Bologna, Bologna, Italy  
{mirko.viroli,a.ricci}@unibo.it

**Abstract.** Recent works in the context of self-organisation foster the idea of engineering large-scale situated systems by taking an aggregate stance: system design and development are better conducted by abstracting away from individuals' details, rather directly engineering (designing, programming, verifying) the overall system behaviour, as if it were executed on top of a single, continuous-like machine. As a consequence, concerns like interaction protocols, self-organisation, adaptation, and large-scaleness, get automatically hidden “under the hood” of the platform supporting aggregate computing, with notable advantages in raising the abstraction level and scaling with behaviour complexity. This paper provides an initial exploration of potentials and challenges of using aggregate computing techniques in the context of multi-agent systems, considering impact on large-scale reactive MASs, environment engineering and its cognitive exploitation, and on collective team-work by the notion of aggregate plan.

## 1 Introduction

Self-organisation mechanisms support adaptivity and resilience in complex natural systems at all levels, from molecules and cells to animals, species, and entire ecosystems [25]. A long-standing aim in computer science is to find effective engineering methods for exploiting such mechanisms to bring similar adaptivity and resilience to a wide variety of complex, large-scale computing applications—in smart mobility, crowd engineering, swarm robotics, etc. Practical adoption, however, poses serious challenges, since self-organisation mechanisms often trade efficiency for resilience, and are often difficult to predictably compose to meet more complex specifications.

On the one hand, in the context of multi-agent systems (MASs), self-organisation is achieved relying on a weak notion of agency: following a biology inspiration, agents execute simple and pre-defined behaviour, out of which self-organisation is achieved by emergence [12]—ant foraging being a classical example. This approach however hardly applies to open and dynamic contexts in which what is the actual behaviour to be carried on by a group of agents is to be decided (or even synthesised) at run-time: offline fine-tuning of system parameters often hampers applicability to real-life, non trivial applications.

On the other hand, a promising set of results towards addressing solid engineering of open self-organising systems are being achieved under the umbrella of *aggregate programming* [3]. Its main idea is to shift the focus of system programming from the individual’s viewpoint to the aggregate viewpoint: one no longer programs the single entity’s computational and interactive behaviour, but rather programs the collection. This is achieved by abstracting away from the discrete nature of computational networks, by assuming that the overall executing “machine” is a sort of (space-time) computational continuum able to manipulate distributed data structures: actual self-organisation mechanisms sit below, and are they key for automatically turning aggregate specifications into individual behaviour. Aggregate programming is grounded in the computational field calculus [9], its incarnation in the Protelis programming language [19], on studies focussing on formal assessment of resiliency properties [23], and building blocks and libraries built on top to support applications in the context of large scale situated systems [2].

This paper aims at analysing the potentials and challenges that can arise when combining techniques of aggregate programming in the context of MASs. In Sect. 2 we start recapping the main elements of aggregate computing. Section 3 depicts a methodology for engineering large-scale reactive MASs on top of aggregate computing, based on the construction of layers of resilient composable functions, raising the abstraction level to address system complexity. Section 4 discusses impact on environment engineering: aggregate computing is about manipulation of computational fields [9, 14], which can be seen as distributed “traces” or “stigma” that agents leave in the spatial environment as a coordination tool, up to be exploited to externalise true fields of beliefs, goals, and intentions. Section 5 presents early ideas on applying aggregate computing to ground a notion of “aggregate plan”, a collective plan shared and cooperatively executed by a dynamic team of agents, developed so as to abstract from participants’ number and details. Section 6 concludes providing final remarks.

## 2 Aggregate Programming

Most paradigms of distributed systems development, there including the multi-agent system approach, are based on the idea of programming each single individual of the system, in terms of its computational behaviour (goals, plans, algorithm, interaction protocol), typically considering a finite number of “roles”, i.e., individual classes. This approach is argued to be problematic: it makes it complicated to reason in terms of the effect of composing behaviours, and it forces the programmer to mix different concerns of resiliency and coordination—using middlewares that externalise coordination/social abstractions and interaction mechanisms only partially alleviates the problem [5, 24].

These limits are widely recognised, and motivated work toward aggregate programming across a variety of different domains, as surveyed in [1]. Historically such works addressed different facets of the problem: making device interaction implicit (e.g., TOTA [14]), providing means to compose geometric and topological constructions (e.g., Origami Shape Language [16]), providing means for

summarising from space-time regions of the environment and streaming these summaries to other regions (e.g., TinyDB [13]), automatically splitting computational behaviour for cloud-style execution (e.g., MapReduce [10]), and providing generalisable constructs for space-time computing (e.g., Proto [15]).

Aggregate computing, based on the field calculus computational model [9] and its embodiment in Protelis programming language [19], lies on top of the above approaches and attempts a generalisation starting from the works on space-time computing, which are explicitly designed for distributed operation in a physical environment filled with embedded devices, but can be extended to work on arbitrary physical/logical environments.

## 2.1 Computing at the Aggregate Level

The whole approach of aggregate computing starts from the observation that the complexity of large-scale situated systems must be properly hidden “under-the-hood” of the programming model, so that composability of collective behaviour can be more easily supported and better address the construction of complex systems. Aggregate programming is then based on the following three principles:

1. The “machine” being programmed is a region of the computational environment whose specific details are abstracted away (perhaps even to a pure spatio-temporal continuum);
2. The program is specified as a manipulation of data structures with spatial and temporal extent across that region;
3. These manipulations are actually carried out in a robust and self-organising manner by the aggregate of cooperating devices situated in that region, using local interactions.

As an example, consider the problem of designing crowd safety services based on peer-to-peer interactions between crowd members’ smart-phones. In this example, smart-phones could interact to collectively estimate the density and distribution of crowding, seen as a distributed data structure mapping each point of space to a real-value indicating the crowd estimation, namely, a *computational field* (or simply *field*) of reals [9, 14]. This can be in turn used as input for several other services: warning systems for people nearby dense regions (producing a field of booleans holding true where warning has to be set), dispersal systems to avoid present or future congestion (producing a field of directions suggested to people via their smartphones), steering services to reach points-of-interest (POI) avoiding crowded areas (producing a field of pairs of direction and POI name). Building such services in a fully-distributed and resilient way is very difficult, as it comes to achieve self-\* behaviour by careful design of each device’s interaction with its neighbours. With aggregate programming, on the other hand, one instead naturally reasons in terms of an incremental construction of computational fields, with the programming platform taking care of turning aggregate programs into programs for the single device.

## 2.2 Constructs

The *field calculus* [9] captures the key ingredients of aggregate neighbour-based computation into a tiny language suitable for grounding programming and reasoning about correctness – recent works addressed type soundness [9] and self-stabilisation [23] – and is then incarnated into a Java-oriented language called Protelis [19], which we here use for explanation purposes. The unifying abstraction is that of computational field, and every computation (atomic or composite) is about functionally creating fields out of fields. Hence, a program is made of an expression  $e$  to be evaluate in space-time (ideally, in a continuum space-time, practically, in asynchronous rounds in each device of the network) and thus producing a field “evolution”. Four mechanisms are defined to hierarchically compose expressions out of values and variables, each providing a possible syntactic structure for  $e$ :

- **Application:**  $\lambda(e_1, \dots, e_n)$  applies “functional value”  $\lambda$  to arguments  $e_1, \dots, e_n$  (using call-by-value semantics).  $\lambda$  can either be a “built-in” primitive (any non-aggregate operation to be executed locally, like mathematical, logical, or algorithmic functions, or calls to sensors and actuators), a user-defined function (that encapsulates reusable behaviour), or an anonymous function value  $(x_1, \dots, x_n) \rightarrow e$  (possibly passed also as argument, and ultimately, spread to neighbours to achieve open models of code deployment [9])—in the latter case Protelis ad-hoc syntax is  `$\lambda$ .apply( $e_1, \dots, e_n$ )`.
- **Dynamics:** `rep( $x \leftarrow v$ ){ $e$ }` defines a local state variable  $x$  initialised with value  $v$  and updated at each computation round with the result of evaluating the update expression  $e$ .
- **Interaction:** `nbr( $e$ )` gathers by observation a map at each neighbour to its latest resulting value of evaluating  $e$ . A special set of built-in “hood” functions can then be used to summarise such maps back to ordinary expressions, e.g., `minHood( $m$ )` finds the minimum value in the map  $m$ .
- **Restriction:** `if( $e$ ){ $e_1$ }else{ $e_2$ }` implements branching by partitioning the network into two regions: where  $e$  evaluates to true  $e_1$  is evaluated, elsewhere  $e_2$  is evaluated. Notably, because `if` is implemented by partition, the expressions in the two branches are encapsulated and no action taken by them can have effects outside of the partition.

A simple example using the various constructs (colouring field calculus keywords magenta, built-in functions green, user-defined functions red, and variables green) is:

```
def distance-avoiding-obstacle (source, obstacle){
  if(obstacle) {infinity} else {
    rep(d<-infinity) {
      mux(source, 0, minHood+(nbrRange + nbr(d)))
    } } }
```

This code creates a field of estimated distances to devices where `source` is `true`, using a metric that computes such distances by circumventing devices where `obstacle` is `true`. In the region outside the obstacle (by `if`), a distance estimate `d` (established by `rep`) is computed using built-in selector `mux` to set sources to 0 and other devices by the triangle inequality, taking the minimum value obtained by adding the distance to each neighbour to its estimate of `d` (by `nbr`).

### 3 Impact on Building Large-Scale Self-Organising MASs

Aggregate computing makes weak assumptions on the underlying computing platforms, that well match those of large-scale reactive MASs: asynchronous agent computation, broadcast of messages to neighbours, perception/action on the local part of the physical environment. This paves the way for using aggregate computing techniques for developing large-scale self-organising MASs.

#### 3.1 Raising the Abstraction Level

While the constructs of aggregate computing form an universal set, they are also too low level to be readily used for building complex distributed services like self-organising MASs. To raise the level of abstraction it is fruitful to identify a collection of general combinators (or “building blocks”), which encapsulate reusable coordination mechanisms, and allow one to bypass the trickier aspects of field calculus. Such combinators set is formed by careful selection of coordination mechanisms needed for complex situated MASs, and hence should be (i) self-stabilising, meaning that they reactively adjust to changes in environment, (ii) scalable to large MASs, and (iii) preserve these resilience properties when composed together into more complex coordination services.

Some operators have been identified already, in [2]. Two of them seem particularly relevant for the context of MASs: new operators `G` and `C`, to be used along with constructs `if` and built-ins. The two building blocks are defined as:

- `G(source,init,metric,accumulate)` is a “spreading” operation generalising distance measurement, broadcast, and projection. It may be thought of as executing two tasks: it computes a field of shortest-path distances from a `source` region (indicated as a Boolean field) according to the supplied function `metric`, then propagates values along the gradient of the distance field away from source, beginning with value `initial` and accumulating along the gradient with `accumulate`.
- `C(potential,accumulate,local,null)` is complementary to `G`, accumulating information to the `source` down the gradient of a supplied `potential` field. Beginning with an idempotent `null`, at each device, the `local` value is combined with “uphill” values using a commutative and associative function `accumulate`, to produce a cumulative value at each device in the `source`.

Although there are only a few operators, they are so general as to cover, individually or in combination, a large number of the common coordination patterns used in design of resilient systems. With appropriate implementation in field calculus, this system of operators can thereby provide an expressive programming environment that provides strong guarantees of resilience and scalability, as established in [2].

### 3.2 Towards Libraries of Collective Distributed Sensing and Action

Key operations of large-scale self-organising MASs involve the need of perceiving events distributed in a whole space region, elaborate them, and properly perform an actuation again into a whole space region. Operators **G** and **C** provide a good “lingua franca” for expressing behaviours on top of primitive aggregation/collection operations.

For example, operator **G** (along with built-ins) can generate a number of interesting functions related to distributed action and information diffusion. One such common computation in spatially embedded systems is estimating the distance from one or more designated “source” devices to others nearby, which can be implemented by a simple application of **G**, beginning with zero and using estimated device-to-device distance as a metric:

```
def distanceTo(source) {
  G(source, 0, () -> {nbrRange}, (v) -> {v + nbrRange})
}
```

Likewise, another common coordination action, broadcasting a value across the network from a source, can be implemented by another application of **G**:

```
def broadcast(source, value) {
  G(source, value, () -> {nbrRange}, (v) -> {v})
}
```

Other **G**-based operations include construction of a Voronoi partition and a “path forecast” that marks paths that cross an obstacle or region of interest.

Similarly, operator **C** enables functions related to information perception, such as accumulating the sum of all the values of a variable in a region

```
def summarize(sink, accumulate, local, null) {
  C(distanceTo(sink), accumulate, local, null)
}
```

or computing the variable’s average or maximum value in that region.

Just as when building any other software library, these API functions can be combined together to create higher level libraries. For example, an average function shared throughout a region can be implemented by applying **broadcast** to the output of **summarize**, as follows:

```
def average(sink,value){
  broadcast(sink, summarize(sink,+,value,0) / summarize(sink,+,1,0))
}
```

### 3.3 Challenges

The main research challenges we identify to foster exploitation of aggregate computing for building large-scale reactive MASs include:

- Extracting from various application contexts general building blocks and APIs to help development of real-life complex systems;
- Designing a platform support for MASs based on aggregate computing, where purely local interactions and cloud-based communications can be dynamically combined;
- Integrating field calculus constructs into agent languages (such as Jason), to streamline combination with existing agent development methodology.

## 4 Impact on Building MAS Environment

Turning our attention to a stronger notion of agency, how can aggregate programming affect the agent-oriented abstractions rooting MAS engineering? An effective way to do this is by means of the notion of environment as a first-class design and programming abstraction [21, 24].

### 4.1 Coordination Artifacts Enacting Computational Fields

The infrastructural substrate that reifies computational fields, which we can call the *computational fields fabric*, can be modelled as the application environment where agents are logically situated, encapsulating the functionalities that agents can exploit to perform their individual and global tasks. In particular, the computational fields fabric can be characterised as a distributed *coordination artifact* [18], since it can be exploited by agents for coordination and self-organisation purposes. Field calculus and Protelis are the basic tools on top of which we can program such a distributed coordination medium (like in the case of programmable coordination media [11]), making it possible to define the coordination and self-organisation functionalities in a declarative and macro way on the one hand, and execute it in a fully decentralised way on the other hand.

As an example, it can be programmed so as to create a gradient field (with **G** operator), so that agents willing to advertise an event can inject information in the environment locally, which gets then distributed around, and can be exploited by other agents perceiving the environment, either to just observe information or to move towards its source.

This view generally allows for conceiving in a clean way systems where the environment encapsulates functionalities useful for self-organisation and collective adaptation, still retaining agent full autonomy.

## 4.2 Cognitive Fields

The integration of aggregate programming and agents lead to consider quite naturally the opportunity of exploiting aggregate coordination functionalities by cognitive agents too, i.e., thinking about computational fields designed in terms of cognitive agents' mental attitudes, such as beliefs and goals. In other words, with cognitive agents, a computational field would represent a kind of distributed, decentralised, and externalised *mental state*, which evolves according to the agent actions and the rules of field evolution specified in the environment program. In that perspective, we envision some strong connection with our previous works exploring the notion of *cognitive stigmergy* [20] and with contributions in the cognitive science literature discussing the idea of the environment as *extended mind* [8]. It is possible to consider three different levels of cognitive fields:

- *Belief fields* – Belief fields are the simplest case, in which a computational field is like a classic partially observable environment whose percepts are modelled as beliefs by the agent situated in it. The aggregate program in this case specifies how some belief should be distributed among the agents depending on their position inside the field. More generally, the aggregate program defines the rule by which the overall distributed “belief” state can evolve and be influenced by each agent and by the environment. As an example, distributed aggregation and then diffusion of information perceived by temperature sensors can be used to automatically create a constant field of the average temperature value across a region of space, which can be interpreted and used as a belief field by the MAS.
- *Goal fields* – In a goal field, the values manipulated by the field are the goals that are meant to be adopted by the agents that are located in some position of the environment. Thus, the aggregate program in this case specifies a *division of labor*, or how tasks are meant to be allocated to agents. For instance, operator **G** can be used to create a Voronoi partition, dividing the overall space into a set of regions based on proximity to a set of  $n$  source agents. Each such agent  $n_i$  can be considered as initiator of a distributed goal  $g_i$ , diffused to all agents in the region created by  $n_i$ . The resulting partition field is hence seen as a goal field, allocating  $n$  goals to the MAS.
- *Intention fields* – In an intention field, the values manipulated by the field are the intentions that the agent located in some position of the field has, namely, actions to execute to behave collectively. Thus, the aggregate program in this case specifies a spatial-dependent concept of task. For instance, to steer people towards a POI in a complex pervasive environment, one could establish a gradient field from the POI, on top of which a field of directions towards the source can be created. This can be understood as a field of intentions, feeding e.g. pervasive displays that will use a direction to show a direction sign.

## 4.3 Tooling

From a technological point of view, enacting computational fields by environment artifacts makes it possible to exploit existing environment-based technologies to



integrate aggregate programming with existing MAS programming tools. Main examples are EIS [4] and CArTAgO [21].

In the latter case in particular, we can design a set of *artifacts* [17, 21] that make it possible for an agent to perceive and act upon a computational field, as well as to manage the set of computational fields, creating new ones or disposing existing ones. More in detail, in order to work within a computational field, an agent can be equipped with an artifact conceptually representing a piece of field, making it observable (by means of observable properties) both the value of the field in the agent position as well as the values of the neighbourhood. By exploiting CArTAgO with cognitive agent programming languages – such as in the case of JaCaMo [6] based on the Jason agent programming language [7] – this modelling makes it possible to directly implement belief fields, since artifact observable properties are mapped into beliefs of agents observing the artifact. Goal and intention fields can be implemented by using observable properties to represent goals and intentions managed by the field. In this case, using Jason for instance, agents can be equipped with suitable plans to react to changes to the beliefs mapping these observable properties so as to e.g. adopting new goals or adding new plans to the plan library, according to the need.

In this framework, Protelis could be used as high-level language to program the single artifact, to be properly compiled to feed CArTAgO.

#### 4.4 Challenges

Among the many research challenges spawning from the idea of aggregate computing as an environment process, we identify:

- Develop suitable models and infrastructures to support flexible computational fields by environment abstractions;
- Extend the notion of cognitive stigmergy to deal with spatially distributed computational fields;
- Study the consequence of aggregate agent reasoning, in theory, models and implementations of intelligent systems.

### 5 Impact on Aggregate Plans

Another fruitful idea for the integration between aggregate computing and MASs is that of considering an aggregate program as “an aggregate plan”, which an agent can either create or receive from peers, and can deliberate to execute or not in different moments of time.

#### 5.1 Life-Cycle of Aggregate Plans

In our model, aggregate plans are expressed by anonymous functions of the kind  $() \rightarrow e$ , where  $e$  is a field expression possibly calling API functions available as part of each agent’s library. One such plan can be created in two different

ways, by suitable functions (whose detail we abstract away): first, it can be a sensor `sns-injected-function` to model the plan being generated by the external world (i.e. a system programmer) and dynamically deployed; second, it can model a local planner `plan-creation` that synthesises a suitable plan for the situation at hand. When the plan is created, it should then be shared with other agents, typically by a broadcasting pattern—the full power of field calculus can be used to rely on sophisticated techniques for constraining the target area of broadcasting.

Agents are to be programmed with just a minimal `virtual-machine`-like code [9] that makes it participate to this broadcast pattern, so as to receive all plans produced remotely in the form of a field of pairs of a description of the plan and its implementation by the anonymous function. Among the plans currently available, by the restriction operator `if` the agent can autonomously decide which one to actually execute, using as condition the result of a built-in deliberation function that has access to the plan’s description.

Note that if/when an aggregate plan is in execution, it will make the agent cooperatively work with all the other agents that are equally executing the same aggregate plan. This “dynamic team” will then coherently bring about the social goal that this plan is meant to achieve, typically expressed in terms of a final distributed data structure, used as input for other processes or to feed actuators (i.e., to make agents/devices move). The inner mechanisms of aggregate computing smoothly support entering/quitting the team, making overall behaviour spontaneously self-organise to such dynamism.

## 5.2 Mapping Constructs, and Libraries

As a plan is in execution, the operations of aggregate programming that it includes can be naturally understood as “instructions” for the single agent, as follows:

- Function application amounts to any pure computation an agent has to execute, there including algorithmic, deliberation, scheduling and planning activities, as well as local action and perception.
- Repetition construct is instead used to make some local result of execution of the aggregate plan persist over time, e.g. modelling belief update.
- Neighbour field construction is the mechanism by which information about neighbour agents executing the same plan can be observed, supporting the cooperation needed to make the plan be considered as an aggregate one.
- Restriction can be used inside a plan to temporarily structure the plan in sub-plans, allowing each agent to decide which of them should be executed, i.e., which sub-team has to be dynamically joined.

As explained in Sect. 3, one of the assets of aggregate programming is its ability of defining libraries of reusable components of collective behaviour, with formally provable resilience properties. Seen in the context of agent programming, such libraries can be used as libraries of reusable aggregate plans, built on top of building blocks:

- Building block **G** is at the basis of libraries of “distributed action”, namely, cooperative behaviour aimed at acting over the environment or sets of agents in a distributed way.
- Building block **C** conversely supports libraries of “distributed perception”, namely, cooperative behaviour aimed at perceiving the environment or information about a set of agents in a distributed way.
- The combination of building blocks **G** and **C**, and others [2], allows one to define more complex elements of collective adaptive behaviour, generally used to intercept distributed events and situations, compute/plan response actions, and actuate them collectively.

### 5.3 Challenges

The notion of aggregate plan suggests several research directions, with the goal of addressing the following challenges:

- study planning techniques for the dynamic creation of aggregate plans;
- experiment the pragmatics of aggregate plans, to explore their abilities of supporting smooth, self-adaptive entering and quitting from the team playing an aggregate plan;
- devise new linguistic constructs for the field calculus to empower its applicability of model for aggregate plans.

## 6 Conclusions

Aggregate computing is a new metaphor for building distributed systems, with notable impact to the engineering of “complexity”, thanks to its ability to: *(i)* reason in term of field calculus programs to formally derive its behavioural properties [23]; *(ii)* create reusable combinators of wide applicability, to raise the abstraction layer of system development [2]; *(iii)* promote a methodology for substitutability of components to improve performance [22]; and *(iv)* address the problem of platform support in a rather abstract way so as to smoothly support different computation/communication models. All this features are seemingly key for MASs as well.

On the other hand, aggregate programming has also the potential of deeply affecting some aspects of agent theory, fostering a more deep understanding of how “computational fields” can be perceived and exploited by cognitive agents. This can shed light to new methodologies for building intelligent distributed systems, where availability of a huge number of agents can turn from a serious coordination problem to an opportunity for building effective, efficient and resilient systems.

Ultimately, aggregate computing and MASs have the potential of combining into a new powerful notion of “agent aggregate”, which this paper only started exploring in its many facets, and which will be matter of our future research investigations.

## References

1. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: languages for spatial computing. In: Mernik, M. (ed.) *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pp. 436–501. IGI Global, Hershey (2013). <http://arxiv.org/abs/1202.5509>
2. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: *Workshop on Foundations of Complex Adaptive Systems (FOCAS)* (2014)
3. Beal, J., Viroli, M.: Space–time programming. *Philos. Trans. R. Soc. Lond A: Math. Phys. Eng. Sci.* **373**, 2015 (2046)
4. Behrens, T., Hindriks, K., Dix, J.: Towards an environment interface standard for agent platforms. *Ann. Math. Artif. Intell.* **61**(4), 261–295 (2011)
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. *Sci. Comput. Programm.* **78**(6), 747–761 (2013)
6. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. *Sci. Comput. Programm.* **78**(6), 747–761 (2013)
7. Bordini, R.H., Hübner, J.F., Wooldrige, M.: *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. Wiley, Hoboken (2007)
8. Clark, A., Chalmers, D.: The extended mind. *Analysis* **58**(1), 7–19 (1998)
9. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) *Formal Techniques for Distributed Objects, Components, and Systems. LNCS*, vol. 9039, pp. 113–128. Springer, Heidelberg (2015)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
11. Denti, E., Natali, A., Omicini, A.: Programmable coordination media. In: Garlan, D., Le Métayer, D. (eds.) *COORDINATION 1997. LNCS*, vol. 1282, pp. 274–288. Springer, Heidelberg (1997)
12. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. *Nat. Comput.* **12**(1), 43–67 (2013)
13. Madden, S.R., Szewczyk, R., Franklin, M.J., Culler, D.: Supporting aggregate queries over ad-hoc wireless sensor networks. In: *Workshop on Mobile Computing and Systems Applications* (2002)
14. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the tota approach. *ACM Trans. Softw. Eng. Methodol.* **18**(4), 1–56 (2009)
15. MIT Proto. Software available at <http://proto.bbn.com/>. Accessed 1 January 2012
16. Nagpal, R.: *Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics*. Ph.D. thesis, MIT (2001)
17. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. *Auton. Agent. Multi-Agent Syst.* **17**(3), 432–456 (2008)
18. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: environment-based coordination for intelligent agents. In: Jennings, N.R., Sierra, C., Sonenberg, L., Tambe, M. (eds.) *Proceedings of AAMAS 2004*, vol. 1, pp. 286–293. ACM, 19–23 July 2004
19. Pianini, D., Beal, J., Viroli, M.: Practical aggregate programming with protelis. In: *ACM Symposium on Applied Computing (SAC 2015)* (2015) (To appear)

20. Ricci, A., Omicini, A., Viroli, M., Gardelli, L., Oliva, E.: Cognitive stigmergy: towards a framework based on agents and artifacts. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 124–140. Springer, Heidelberg (2007)
21. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. *Auton. Agents Multi-Agent Syst.* **23**(2), 158–192 (2011)
22. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2015) (2015)
23. Viroli, M., Damiani, F.: A calculus of self-stabilising computational fields. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 163–178. Springer, Heidelberg (2014)
24. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi-Agent Syst.* **14**(1), 5–30 (2007)
25. Zambonelli, F., Viroli, M.: A survey on nature-inspired metaphors for pervasive service ecosystems. *Int. J. Pervasive Comput. Commun.* **7**(3), 186–204 (2011)

Engineering Multi-Agent Systems

Third International Workshop, EMAS 2015, Istanbul,  
Turkey, May 5, 2015, Revised, Selected, and Invited  
Papers

Baldoni, M.; Baresi, L.; Dastani, M. (Eds.)

2015, X, 231 p. 55 illus. in color., Softcover

ISBN: 978-3-319-26183-6