Journal of
Sensor and Actuator Networks

MDPI

Article

A Programming Approach to Collective Autonomy

Roberto Casadei *, Gianluca Aguzzi and Mirko Viroli

Alma Mater Studiorum–Università di Bologna, Cesena, Italy; gianluca.aguzzi@unibo.it (G.A.); mirko.viroli@unibo.it (M.V.)

* Correspondence: roby.casadei@unibo.it

Abstract: Research and technology developments on autonomous agents and autonomic computing promote a vision of artificial systems that are able to resiliently manage themselves and autonomously deal with issues at runtime in dynamic environments. Indeed, autonomy can be leveraged to unburden humans from mundane tasks (cf. driving and autonomous vehicles), from the risk of operating in unknown or perilous environments (cf. rescue scenarios), or to support timely decision-making in complex settings (cf. data-centre operations). Beyond the results that individual autonomous agents can carry out, a further opportunity lies in the collaboration of multiple agents or robots. Emerging macro-paradigms provide an approach to programming whole collectives towards global goals. Aggregate computing is one such paradigm, formally grounded in a calculus of computational fields enabling functional composition of collective behaviours that could be proved, under certain technical conditions, to be self-stabilising. In this work, we address the concept of collective autonomy, i.e., the form of autonomy that applies at the level of a group of individuals. As a contribution, we define an agent control architecture for aggregate multi-agent systems, discuss how the aggregate computing framework relates to both individual and collective autonomy, and show how it can be used to program collective autonomous behaviour. We exemplify the concepts through a simulated case study, and outline a research roadmap towards reliable aggregate autonomy.

Keywords: collective autonomy; self-organisation; aggregate computing; multi-agent systems; coordination

Citation: Casadei, R.; Aguzzi, G.; Viroli, M. A Programming Approach to Collective Autonomy. *J. Sens. Actuator Netw.* **2021**, *1*, 1. https://doi.org/

Academic Editor: Firstname Lastname

Received:
Accepted:
Published:

1. Introduction

Research and technology trends promote a vision of artificial systems that are able to resiliently manage themselves and autonomously deal with issues at runtime in dynamic environments. Such a vision is mainly investigated by two related research threads. One is the field of autonomic computing [1] and self-adaptive systems [2], which promote the development of Information and Communications Technology (ICT) systems able to self-manage given a set of high-level goals. Indeed, endowing systems with higher degrees of autonomy can be leveraged to unburden humans from mundane tasks (cf. driving and autonomous vehicles), from the risk of operating in unknown or perilous environments (cf. rescue scenarios), or to support timely decision-making in complex settings (cf. data-centre operations). The other is the field of multi-agent systems (MAS) [3], which evolved from the field of distributed artificial intelligence [4]. An MAS is a system of agents, i.e., a collection of autonomous entities interacting with their environment [5] and other agents to satisfy their design objectives. From an engineering point of view, agents and related abstractions are considered useful tools for the analysis and design of complex software-based systems. There are two key problems in the use and development of agents: the design of individual agents (micro level) and the design of a society of agents (macro level) [3].

Indeed, beyond the results that individual autonomous agents can carry out, a further opportunity lies in the collaboration of multiple agents or robots. Emerging macro-paradigms [6,7] provide an approach to programming whole collectives towards

global goals. Aggregate computing or programming [7,8] is one such paradigm, formally grounded in a calculus of computational fields [9] (maps from agents to values) enabling functional specification and composition of collective behaviours. The major benefit of the aggregate programming approach is that it explicitly addresses collective adaptive behavior, rather than the behavior of individuals (which is addressed indirectly, as a consequence of the intended global behavior). The idea is to code scripts, conceptually executed by the collective as a whole, in terms of reusable building blocks of collective tasks [10] capturing both state, behavior, and interaction, crucially enjoying formal mapping to the behavior of individuals and often provable convergence properties [7,11]. We argue that this programming approach, which originated from the research areas of coordination [12] and spatial computing [6] (see [7] for a historical note), can be suitable to MAS programming. Accordingly, in this work, we address the concept of collective autonomy, i.e., the form of autonomy that applies at the level of a group of individuals. Though this notion has been investigated in the literature (cf. Section 2), there are mainly preliminary approaches for practical programming of collective autonomous behavior by a global perspective; so, in this manuscript, we address this software engineering problem explicitly, and sketch a roadmap for further research. Therefore:

- we provide a review of literature about autonomy and especially collective autonomy in MASs (Section 2);
- we analyze the aggregate computing framework by the perspective of autonomy, by covering its positioning with respect to individual and collective autonomy, and showing how it can support adjustable autonomy (Section 3);
- we exemplify the discussion through a simulated case study, investigating (i) the relationship between individual goals/autonomy and collective goals/autonomy; and (ii) the relationship between structures and collective autonomy (Section 4); and
- we discuss gaps in the literature on programming reliable collective autonomy and delineate a research roadmap (Section 5).

Finally, we conclude with a wrap-up in Section 6. In summary, in this route, we cover how to aggregate computing relates to and supports various forms of autonomy, and propose it as a framework for programming and simulating collective autonomous behavior, in a way that differs from related works and that opens up various research directions regarding actionable notions of collective autonomy.

2. Background and Related Work

This section provides the background for our contribution, which is positioned at the intersection of distributed artificial intelligence and software engineering.

2.1. Autonomy in Software Engineering and Multi-Agent Systems

At a first level, autonomy is used as a general informal notion, a characteristic assumed to be possessed by some entities during design. Etymologically, autonomy refers to an entity that follows its own laws, i.e., that is able to self-regulate its behavior. At a second level, concrete and (semi-)formal notions of autonomy are developed to support engineering tasks under specific viewpoints—see Table 1 for a summary. For instance, from a programming language perspective, agents are computationally autonomous in the sense that they "encapsulate invocation" (i.e., they act as internally defined, e.g., by rules or goals). Therefore, agents can be thought of as the next step of an evolution from monoliths to modules (encapsulation of behavior), to objects (encapsulation of state in addition to behavior), to active objects/actors (decoupling invocation from execution) [13].

Autonomy, together with agency (the ability to act), appears to one of the key defining and agreed upon characteristics of agents in MAS research [14]. From autonomy and agency, other features naturally arise. Agents are proactive: they are not only reactive to external stimuli, but driven towards action by an inner force. Agents are social and interactive, as autonomy makes sense in a relational context such as a society (MAS). The

importance of interaction has motivated research on how to effectively rule it to promote the satisfaction of design goals—the field of coordination [12].

As a relational notion, it is more precise to say that an agent is autonomous *(i) from something* and possibly *(ii) with respect to something* [15]. Accordingly, it is possible to distinguish between social autonomy (the autonomy of an agent from other agents) and non-social, environmental autonomy (the autonomy of an agent from the environment). It is important to remember that autonomy is a gradable notion from the extremes of no autonomy to full or absolute autonomy. Regarding the object for which autonomy is considered, researchers typically distinguish between agents that are plan-autonomous (i.e., are free to determine the course of actions to reach given goals) and agents that are goal-autonomous (i.e., are free to determine their own goals). These forms are also called as executive and motivational autonomy [16], respectively. Another common distinction is between weak and strong agency. In the latter, goals are explicitly represented. An approach to (strong) agency is to consider agents as intentional entities with mental states such as epistemic (e.g., percepts, beliefs) and motivational (e.g., desires, intentions) states. A well-known model in this class is the Belief–Desire–Intention (BDI) control architecture, which counts several implementations and variants [17].

Table 1. A summary of common notions of autonomy.

Dimension	**Elements/Terms**	
Reference entity	Agent	Group of agents
	individual autonomy	collective autonomy
Autonomy "from" something	Other agents	Environment
	social autonomy	non-social/environmental autonomy
Autonomy "with respect to" something	Goals	Plans
	motivational autonomy	executive autonomy
Autonomy extremes	None	Full
	no autonomy (passivity)	absolute autonomy (freedom)
Autonomy flexibility	None	Full
	fixed autonomy	adjustable autonomy
Source of autonomy with respect to a component of a reference entity	Internal	External
	endogenous autonomy	exogenous autonomy

2.2. Collective Autonomy

The notion of collective autonomy emerges when the reference agent is not an atomically individual agent but a whole collective, i.e., a collection of individuals (agents or other, possibly non-autonomous agents). As autonomy as a concept tends to be related to the existence (and possibly awareness) of a "self" [18], an autonomous collective tends to be and work as a "unit". Working as a unit requires the components of a collective to be jointly directed towards goals, states of affairs, or values—a concept known as collective intentionality [19,20]. In [19], an (intentional) collective is defined as a collection of agents held together by a "plan", which specifies a "goal" and their "roles". In [21], a formal analysis of collective autonomy is provided. A collective (agent) is defined as a collection of complementary agents sharing a common, collective goal (which, in a sense, reduces the individual autonomy of the members). As for individuals, autonomy for collectives is a gradable and relational notion. A collective may be defined as plan-autonomous (goal-autonomous) if no other entity (internal or external) can change its plans (goals).

The focus on autonomy, together with a collective stance, can be used to model or engineer complex system behavior. This is the idea of autonomy-oriented computing [22], where computation is defined in terms of local autonomous entities, spontaneously (inter-)acting together and with the environment to achieve self-organizing behavior. In the following section, we cover a state-of-the-art programming model for this paradigm.

2.3. Multi-Agent Systems Programming

A recent survey on agent-based programming is by Mao et al. [23]. They classify agent programming languages into three families according to the level they address: individual agent programming (micro-level), agents integration and interaction programming (meso-level), and multi-agent organisation programming (macro-level). Representatives of these classes include cognitive-oriented languages such as AgentSpeak(L)/Jason [24], agent communication and environment modelling languages like KQML [25], CArtAgO [26], and SARL [27], and organisation-oriented programming such as MOISE [28]. Another recent survey by Cardoso et al. [29] distinguishes between general-purpose agent programming languages and languages for agent-based modeling and simulation. In the following, we cover a programming model arising from research on spatial computing [6,30] and field-based coordination [7,31], which provides an original approach to MAS programming that allows driving micro-level activity based on specifications addressing the meso- and macro-levels of a MAS.

2.3.1. Aggregate Programming

Aggregate programming is an approach to specify the collective adaptive or self-organising behaviour of a MAS by a global perspective. The individual behavior of the agents derives from an aggregate program that is conceptually executed by the system as a whole. The aggregate program provides a way to map the local observations of an individual agent (i.e., sensing information, current agent state, and inbound messages from neighbors) to (eventually) globally-coherent local actions (i.e., actuation instructions, and outbound messages). Therefore, an aggregate program covers the aspects of sensing, actuation, computation, and communication to define how the MAS should collectively behave. In particular, we define an aggregate system as a MAS of agents, structurally connected such that an agent can only interact with a subset of other agents known as its neighbors, and repeatedly plays an aggregate program against its up-to-date context (further details about the execution protocol are provided in Section 3.1).

Historically, aggregate programming originated from works drawing inspiration from nature: whereas the biological inspiration led to swarm intelligent MASs, where agent indirectly interact by pheromones [32], the physical inspiration led to the idea of agents acting in environments empowered with potential fields [31]. Recently, aggregate programming has been formally backed by field calculi [7], which provide a compositional approach to global behavior specification based on functions from fields to fields. A (computational) field is a map associating a value to any device of a given domain. So, for instance, controlling the movement of a swarm of drones can be expressed through a field of velocity vectors, which maps any drone of the swarm to a corresponding velocity (speed and direction); the set of low-energy devices can be denoted through a Boolean field holding `true` for devices whose local energy level (as perceived by local sensors, and collectively also denoted as a floating-point field) is under a certain threshold (also a floating-point field). These fields, then, are generally manipulated through three kinds of constructs:

1. Stateful evolution: `rep(init)(f)`—expressing how a field, starting as `init`, should evolve round-by-round through unary function `f`.
2. Neighbour interaction: `nbr(e)`—used to exchange with neighbours the value obtained by evaluating field expression `e`; this locally yields a neighbouring field, i.e., a field that maps any neighbour to the corresponding evaluation of `e`.

3. Domain partitioning: `branch(c){ifTrue}{ifFalse}`—used to partition the domain of devices into two parts: the devices for which field `c` is locally `true`, which evaluate expression `ifTrue`, and those for which `c` yields `false`, which evaluate `ifFalse`.

The idea of aggregate programming is to write programs talking about global behavior (fields) and let these drive the local activity of every device in the system. Aggregate programming is embodied by concrete aggregate programming languages [7], such as ScaFi [33,34], a Domain-Specific Language (DSL) embedded in Scala as well as a toolchain for aggregate system development and simulation [35]. ScaFi is used for the examples in this paper and for the experimental evaluation of Section 4.

We adopt ScaFi in this paper mostly for practical reasons: with respect to other aggregate programming languages such as Proto and Protelis, surveyed in [7], ScaFi is a strongly typed, internal DSL; therefore, it enables straightforward reuse of powerful features from the Scala host language (including its type system, type inference, programming abstractions, libraries) as well as seamless integration with tooling supports for Java Virtual Machine-based languages (including Integrated Development Environments, static analysis, and debugging tools), at the expense of a more constrained syntax and semantics. Additionally, ScaFi also represents an agile framework for testing experimental language features (cf. aggregate processes [34]—referenced in Section 5.1). Hence, among the existing languages for aggregate programming, we believe ScaFi is the one better fitting rich scenarios like those addressed in this paper.

A full account of research about aggregate programming, field calculi, and ScaFi is beyond the scope of this article; the interested reader can refer to [7,34].

Recently, some preliminary work [36] has been carried out to consider the application of the aggregate approach for MAS programming, along with a strong-agency viewpoint. There, two main ideas are proposed. One is the notion of a cognitive field (e.g., fields of beliefs, fields of goals, fields of intentions), which could be used to represent "a kind of distributed, decentralized, and externalized mental state". The other is the notion of an aggregate plan, i.e., a global, collective plan of actions modeling the way in which a dynamic team of agents cooperates towards a social goal in a self-organizing way. In particular, aggregate plans can be created by an initiator agent and iteratively spread by the other agents; any agent has the faculty of choosing to adopt the plan or not; if the plan is adopted, then the agent will execute the corresponding actions, which in general will depend on the agent's position in space and in the team. This mechanism may be suitable where the behavior to be executed is somehow related to the space/environment, where the MAS can be clustered in teams of agents exhibiting uniform behavior, or where the MAS has to embed a decentralized, self-organizing force. A management lifecycle for aggregate plans involve the following phases: synthesis, spreading, collection, selection, execution. In this article, we build on this perspective, and rather focus on the notion of (collective) autonomy.

3. Autonomy in Aggregate Computing

In this section, we analyse the notion of autonomy (cf. Section 2.1) by the aggregate computing and programming perspective (cf. Section 2.3.1). In particular, we propose the aggregate execution protocol as basis of an agent control architecture (Section 3.1), and then discuss aggregate programming of individual (Section 3.2) as well as collective autonomy (Section 3.3).

3.1. Aggregate-Oriented Agent Control Architecture

The field calculus small-step operational semantics [9] provides an abstract aggregate execution protocol for "driving" aggregate behaviour. The aggregate execution protocol typically consists of having an agent repeatedly run (e.g., once per second, or upon change of the local context) a computation round consisting of the following steps:

1. Context evaluation. In this step, the agent looks at its current state, its sensors, and its message box for new information in order to update the local context.

2. Aggregate program evaluation. In this step, the agent runs the aggregate program providing its local context as an input: the evaluation of the program returns an output data structure that can be used for context update.
3. Context action. Using the output of the program evaluation, the agent must
   - update its local state;
   - send a message to neighbours;
   - trigger actuations (if needed).

Such an abstract aggregate execution protocol can effectively be used to define an agent control architecture (Figure 1). Variants of such a control architecture can also be envisaged, e.g., by considering asynchronicity and different rates for context evaluation, computation, and action. Preliminary work supporting this direction can be found in [37,38], where partitioning schemas and programmable schedulers are proposed.

This control architecture also provides a basis for integrating the aggregate paradigm with other agent control architectures, e.g., the cognitive ones based on BDI [17]—which makes for an interesting future fork.

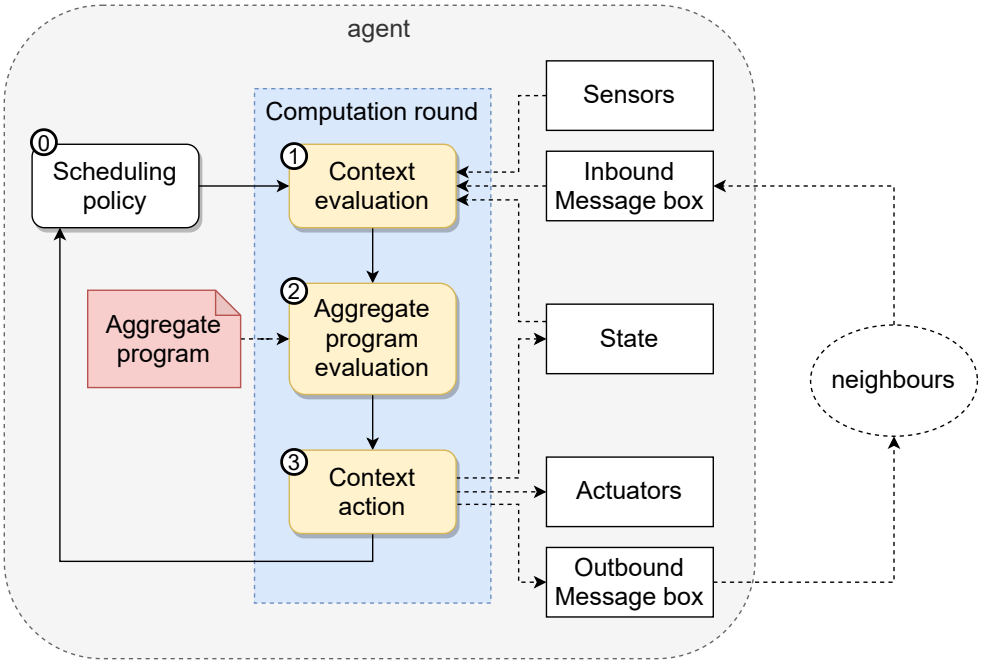
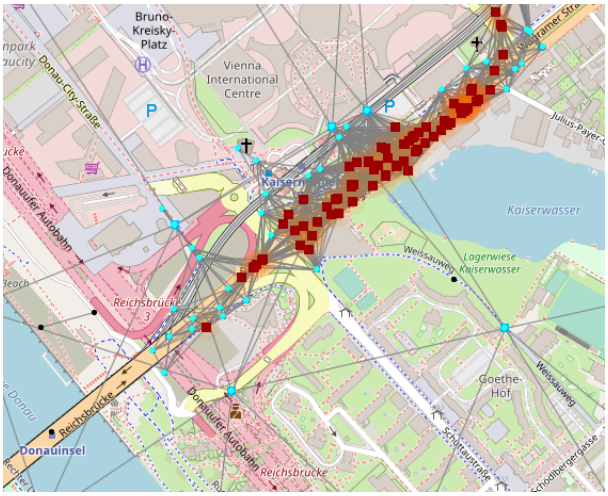


Figure 1. Agent control architecture in aggregate programming. Notation: square boxes denote components; rounded boxes denote activities; dashed rounded boxes denote agents; solid arrows denote control passing; dashed arrows denote data passing.

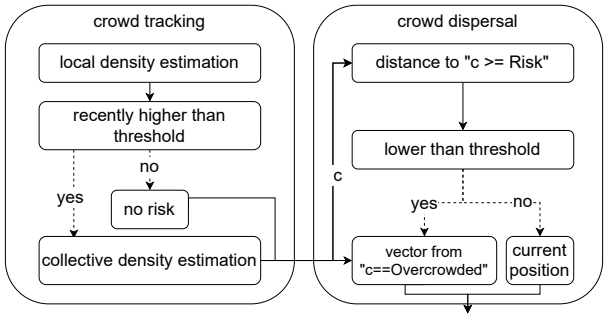
3.2. Individual Autonomy in Aggregate Computing

As a running example, consider a crowd detection and steering application [8] (cf. Figure 2a). When this application is built with the aggregate paradigm, the crowd is represented as an aggregate system consisting of a large-scale, dense network of smartphone- or wearable-augmented people—co-located in a spatial region such as a public exhibition area, a concert, or a stadium. An aggregate program can be continuously played by this system, providing each agent with an estimation of the local density, a local risk level, and possibly—if the vicinity to risky areas exceeds a certain threshold—advice for safe dispersal (in terms of a field of movement directions). In ScaFi, such a program may be implemented, reusing a library of general-purpose aggregate components [10], as per Figure 2c. Function `crowdTracking` (Line 2) runs a collective crowding risk estimation process, by calling `collectiveDensityEstimation` (Line 6), and selects the corresponding output (i.e., a crowding value—`NoRisk`, `Risk`, or `Danger`) only for the devices for which the perceived local density exceeds threshold value `thCrowd`; for the others, the output

is `NoRisk`. Function `collectiveDensityEstimation` (whose code is not shown) may, e.g., using the building blocks in [10], break the system into multiple areas, compute the mean local density in each one of them, and share to all the members the area-wide crowding level (based on whether the average density exceeds threshold `thDanger`). Then, function `crowdDispersal` (Line 10) provides a suggestion for dispersal for all the devices that are closer than `riskRange` from any device with crowding level `Risk`; the suggestion is essentially computed as the opposite of a vector pointing to the center of mass of a close `Overcrowded` device group. We remark that, though the aggregate program is unique (essentially like a shared plan for all the involved agents), its execution is distributed (i.e., decentralized) and local (i.e., with agents interacting with neighbor agents only), hence enabling scalable collective computations.



(**a**)



(**b**)

```
1  /* Shared plans (see also [8]) */
2  def crowdTracking(p: Double, r: Double, w: Double, t: Double,
3                    thCrowd: Double, thDanger: Double): Crowding = {
4    val localDensity = localDensityEstimation(p, r)
5    mux(recentlyTrue(localDensity > thCrowd, t)){
6      collectiveDensityEstimation(p, r, thDanger)
7    } { NoRisk }
8  }
9
10 def crowdDispersal(c: Crowding, r: Double): TargetPosition =
11     // if a device is closer than r to any device with crowding level = Risk or higher
12     // then compute a movement vector to move away from the Overcrowded area, or do not move
13     branch(distanceTo(c >= Risk) < r){ vectorFrom(c == Overcrowded) }{ currentPosition() }
14
15 /* Shared beliefs: see [8] for motivation of the specific values */
16 val p = 0.005 // proportion of people with a corresponding device (agent)
17 val r  = 30   // range in metres for local crowding estimation
18 val w  = 0.25 // fraction of walkable space
19 val t = 60.0  // timeframe (in seconds) for risk monitoring
20 val thCrowd  = 1.08 // relevant crowding threshold
21 val thDanger = 2.17 // dangerous crowding threshold
22 val riskRange = 50  // distance to risk for triggering alert
23
24 /* Aggregate program: main logic */
25 val crowdingLevel = crowdTracking(p, r, w, thCrowd, thDanger, t)
26 crowdDispersal(crowdingLevel, riskRange) // dispersal advice
```

(**c**)

Figure 2. Crowd detection and steering example: snapshot, architecture and program. (**a**) A simulation snapshot of the crowd detection and steering example. Red nodes are devices in an overcrowded area; cyan nodes are nodes at risk since close to overcrowded devices; black nodes are in a safe location. Solid lines denote connectivity links (the longer links are those between infrastructural nodes); (**b**) diagram corresponding to the ScaFi application code below. Boxes denote computational field expressions; solid arrows denote input/output relationships; dashed arrows are a shorthand to denote conditional selection (when initially separated—for `mux`, where the "then" and "else" expressions are both evaluated, but only one is returned) and conditional evaluation (when initially joined—for `branch`, where only one between the "then" and "else" expression is evaluated and returned). (**c**) Aggregate program as implemented in ScaFi.

An aggregate behavior is the result that emerges from the combination of (*i)* an aggregate program; (*ii)* a concrete aggregate execution; (*iii)* an environment—which comprises the behavior of the agents that is out of the control scope of the aggregate program. For the crowd example, the aggregate program expresses how the MAS should determine risky areas and how dispersal processes should be carried out; an aggregate execution may, e.g., set the round frequency to match the levels of mobility; the environment entails elements like spatial distribution and connectivity. Therefore, for an aggregate behavior to actually work, i.e., to correctly carry out its intended functionality (which can be seen as a "social benefit") it is important that the aggregate execution which is tailored to match prefigured features of the environment, is respected by every device, generally. Indeed, there exist guarantees regarding (*i)* self-stabilisation of aggregate computations [11], namely the guarantee to eventually converge to a correct state once perturbations cease; and (*ii)* adaptation of aggregate behaviours to device distribution (density, topology) [39], namely eventual consistency of values, whose approximation improves as the discrete network tends to more densely cover the environment. Under this perspective, an aggregate program may be interpreted as a representation of a social norm, and the aggregate system as a society, or even a normative MAS [40]. So, an agent that does not run the aggregate program is a deviant and not really part of society. On the other hand, playing by the norms implies a limitation of individual autonomy, which is generally traded for a greater, social good. In the crowd example, it is clear that the activity of information gathering and broadcasting is instrumental to achieve good "social" performance in risk detection and dispersal self-correction; moreover, great damage may result from not following dispersal advice—so, the crowd program may be interpreted as the reification of a social norm for safe behavior in a perilous situation. Inspired by normative MAS, deviance can be minimized by leveraging social enforcement mechanisms such as rewards or sanctions [40]. However, an agent of an aggregate MAS may reduce, e.g., the frequency at which rounds are executed, because of a low-energy level, or because its portion of the environment is largely stationary. In other words, the first element of individual autonomy in aggregate MASs revolves around how an agent adheres to the aggregate execution protocol for the application at hand.

The other elements of individual autonomy are those provided by and undergone by the aggregate program. Indeed, an aggregate program may be used to control individual behavior, through local functions containing no aggregate constructs for global coordination. We call this endogenous or delegated autonomy, as it is delegated from the (inside of the) program itself. For instance, the crowd example may be extended to leave some individual autonomy when following the dispersal advice:

```
localDispersalDecision(crowdDispersal(...))
```

where `localDispersalDecision` can be a function potentially different from agent to agent, able to affect the local socially-enforced dispersal direction. This mechanism may also be used to program adjustable autonomy, i.e., the form supporting "agents with graded autonomy properties" [41].

On the other hand, an aggregate program may abstract over or depend on certain elements of agent autonomy. For instance, in the crowd monitoring and control example, the people—and hence the corresponding digital twins (agents)—maintain autonomy regarding mobility (which may also be affected by the crowd itself), since they may choose to follow the dispersal advice or not. We call this exogenous autonomy, as it comes from the outside, beyond the control of the aggregate program. This also includes the influence that an agent may exert on the aggregate program by manipulating its inputs (i.e., state, sensors, and messages). Aggregate programs are typically developed to expressively deal with exogenous autonomous behavior, with strategies that the whole aggregate MAS can follow to adapt and overcome corresponding perturbations to the collectively desired state-of-affairs. Table 2 synthesises the various sources and forms of individual autonomy in aggregate computing.

Table 2. Individual autonomy in aggregate computing.

Individual Autonomy Element	Range (Less–More)
Adherence to the aggregate execution protocol	Fully autonomous (uncooperative)—Completely adherent (control-driven)
Endogenous autonomy (provided by the AC program)	Uncontrolled–Controlled
Exogenous autonomy (undergone by the AC program)	Controlled–Uncontrolled

3.3. Collective Autonomy in Aggregate Computing

As discussed in Section 2.2, collective autonomy is the form of autonomy exhibited by a collective, i.e., a MAS as a whole. Such a concept can be framed around two notions. One is the notion, introduced in this manuscript, of intentional collective stance, which extends the intentional stance [42] to collectives: we may not really know whether the MAS is an intentional collective and what makes it so, but we may still treat it like it was so, e.g., to support reasoning and design activities. For instance, in the crowd example,

```
val crowdingLevel = crowdTracking(p, r, w, thCrowd, thDanger, t)
crowdDispersal(crowdingLevel, riskRange) // dispersal advice
```

the aggregate MAS may be considered as a collective with intentions, i.e., a single distributed entity whose intentions include leaving its internal components free to move at first but also monitoring and ensuring that they do not gather excessively—as a form of (self-)protection.

The second notion is that of joint intentionality [43]. Indeed, collective intentionality requires agents to be jointly directed towards shared activities (plans) or goals. In aggregate computing, the shared goal and the corresponding shared plan to achieve it are reified into an aggregate program (cf. Figure 2c), namely a program that is meant to be played by the whole MAS. Notice that often, like in the crowd example, the individual goal (e.g., moving to a point of interest in the city) may conflict with the collective goal (e.g., moving in the opposite direction to ensure safe dispersal). Therefore, collective autonomy—as the expression of the goals and intentions of an entire collective—potentially reduces individual autonomy, and vice versa (cf. the notion of endogenous or delegated autonomy in the previous section).

Interestingly, an aggregate program does not just embed the collective goals, but also the collective process leading to the selection of collective intentions. For instance, in the crowd example, dispersal is activated once a group of devices determines that the level of danger is sufficiently high (cf. Line 13 in Figure 2c).

As we said, collective autonomy, as a notion, is relational and gradable. The autonomy of a collective can be related to

- the autonomy of the members of the collective—as discussed;
- the environment—namely the extent to which activity depends on environmental situations and events.

Notice that an aggregate MAS can be collectively autonomous even if its overall behavior is highly determined by the deliberation of few individuals—if those individuals have been delegated for decision-making by the collective. Consider the Self-organising Coordination Regions (SCR) pattern [44], which is also exploited in Section 4; a general encoding in ScaFi is as follows:

```
val leaders = S(grain) // sparse-choice: elect leaders at mean distance of grain
val potential = distanceTo(leaders) // gradient field from leaders
val regions = broadcast(potential, mid()) // multi-hop propagation of IDs of leaders
val collectedData = C(potential, membersData()) // collect info towards leaders
val decisions = broadcast(potential, leaderDecision(collectedData)) // propagate leader choices
localAction(decisions)
```

The leader agents (i.e., those for which the `leaders` Boolean field holds `true`) are responsible for making decisions about how the members of the corresponding areas are to behave, but those leaders are elected through a collective process represented by function `S` [45] (where "S" is a contraction of "sparse-choice", i.e., typically a spatially uniform selection of nodes in a situated network). Function `S` aims at selecting leaders at a mean distance of `grain` among them. For the pattern to work, further collaboration is needed among the agents to propagate information. Function `distanceTo(s)` is used to compute a self-healing gradient field [46], i.e., a self-stabilising [11] computational field of minimum distances from any node in the system to the node(s) where `s` is `true`, which is also able to correct the individual estimations by reacting to changes of sources, neighbours, and corresponding positions. Such a gradient field can effectively act as a "`potential` field", namely as a kind of force providing a direction and intensity with respect to a reference point, e.g., for moving information or agents [47]. Indeed, function `broadcast(p,v)` is used to implement a multi-hop information stream of the value `v` at nodes of null potential (i.e., where `p` is 0) outwards by "ascending" the potential field `p`. Dually, function `C(p,v)` (for "collection") provides an information stream converging towards nodes at null potential (i.e., where `p` is 0); sometimes, an operator is provided to specify how the information should be aggregated along the path (e.g., when collecting sets of data, the set union operator may be used to aggregate all the data elements).

We also stress that the collective behavior is not merely the sum of the individual behaviors but the emergent result of repeated individual behaviors involved in a complex network of interactions among related agents (neighbors) and with the environment as well. In other words, an aggregate program provides a schema for self-organizing, self-adaptive behavior that is instantiated once a proper dynamic (defined in terms of a concrete aggregate execution and environmental evolution) is injected over it.

3.4. Summary and Comparison with Related Work

To recap, aggregate programming provides computational mechanisms for supporting various levels of autonomy. Examples include:

- collective autonomy: by cooperative execution of the aggregate program, or in terms of collective structures constraining individual behavior;
- (endogenous) individual autonomy: by calling local functions (as a sort of delegation);
- adjustable autonomy: by using structures or branching mechanisms to regulate the relationship between individuals and collective autonomy, or by controlling the amount of endogenous vs. exogenous autonomy (cf. Table 2).

The discussion in this section is substantiated by the experiments of Section 4 and extended in Section 5 with further considerations on research gaps and potential directions for future investigations.

The proposed approach differs from other works (such as those reviewed in Section 2) in crucial ways. Prominently, it takes a global ("aggregate") perspective on MAS design and programming. Other approaches, such as AgentSpeak(L)/Jason [24], define a MAS by focussing on the behavior of the individual agents, expressed in terms of a number of plans describing how individual goals are to be achieved. However, a Jason program does not directly model collective decision-making or collective action. Note that such perspectives are not alternative but complementary: the support for reasoning available in Jason, based on the BDI architecture, for determining what individual autonomous behavior has to be enacted is not built into aggregate programming. As discussed in Section 5, the combination of cognitive architectures with collective adaptive systems is a theme still to be investigated. The proposed architecture (cf. Figure 1) is simpler than cognitive architectures: it is inspired by self-organising systems [48], and fosters emergence of collective behaviour rather than reasoning.

Works that also consider MASs by a global perspective include organization-oriented programming approaches such as MOISE [28]. MOISE is a very articulated approach for describing agent organisations (i.e., dynamic groups of agents comprising roles, properties,

and interaction protocols) and how agents and organisations interact (e.g., participation to organisations, evolution of organisations, and organisational effects on agents). It is a very rich and flexible model, but it is also quite complex, requiring explicit definitions for the structural, functional, and normative dimensions. By contrast, the aggregate computing approach expresses collective behavior through relatively small scripts (cf. the example in Figure 2c) obtained by composing functions of other collective behavior together. That is, whereas MOISE builds on an explicit representation of organizations, with agents knowing and reasoning about them, aggregate computing favors a more implicit representation of group structures, more typical in the swarm intelligence and self-organisation tradition. As aggregate computing builds on the main abstraction, the computational field, and neighbor-oriented communication, there is low conceptual overhead, with the functional abstraction enabling fine-grained problem decomposition. On the other hand, other approaches of autonomous ensembles programming, such as SCEL [49], do not achieve the same levels of declarativity of aggregate programming languages such as ScaFi. Arguably, the investigation of notions like collective autonomy could take advantage of "simple", compact models which are however able to represent emergent, collective phenomena.

4. Case Study

To showcase the ability of aggregate computing to orchestrate collective behavior comprising both individual and collective autonomy, in this section we present a case study, evaluated by simulation. We conduct the experiments using Alchemist [50], a bio-inspired large-scale multi-agent simulator supporting the ScaFi aggregate programming language [35], which is used to write aggregate programs in our experiments.

Our basic requirements for a simulator include the ability of simulating large-scale networks of mobile devices as well as the ability of defining a dynamics suitable to express the aggregate execution protocol discussed in Section 3.1, which consists of asynchronous rounds of execution with neighborhood-based communication. Therefore, Alchemist represents a first choice as it ships already with a module providing ScaFi support. Moreover, Alchemist is solid and flexible: in the literature, it has been used extensively to simulate multi-agent and aggregate computing systems, in scenarios including crowd simulation [8], drone swarms [34], edge computing [51], and people rescue [36].

The simulations are open-sourced and accessible from a public repository https://github.com/cric96/mdpi-jsan-2020-simulation.

We consider wildlife monitoring as target application domain, which nowadays starts to take advantages of Internet-of-Things (IoT) infrastructure, unmanned ground (UGVs) or aerial (UAVs) vehicles [52], as well as wearables for animals (such as smart collars) or operators [53]. In this scenario, developers could use MAS programming approaches, such as aggregate computing, to coordinate multiple nodes and agents in performing collective activities such as rescuing animals in danger, geofencing, and detecting/tracking poachers. In these experiments, our emphasis is on the specification of autonomous behavior, as well as the emergent relationship between local and collective autonomy. Bridging with realistic environments and data is left as future work.

4.1. Experiment Setup

The collective goal is to find animals in danger and rescue them. The environment consists of a continuous two-dimensional space with an area of 2500 $\times$ 2500 m. There are three types of nodes:

- Animal: an agent that needs to be rescued if it is in danger. The danger status changes are domain-specific dynamic. In this simulation, the animals change their status randomly at a constant rate (one animal every two seconds). It could be a sort of "smart collar".
- Mobile node: an agent that moves around the world and could have the capability to heal an animal.
- Station: a fixed node that works as a gateway for mobile nodes.

In this experiment, Mobile nodes can perform tasks, namely problems to be solved. Our characterization of this concept is a lightweight version of the definition in [54]: each task has a goal, a set of capabilities needed to accomplish it, and spatial constraints (e.g., on the location of the agent or the task). We do not consider the deadline and priority concepts. The tasks that are created collectively (e.g., by the leader on the basis of data collected from other agents) are called Collective tasks. Local tasks are those crafted by the agents themselves based on local perceptions and intentions. We have identified two types of tasks:

- ExploreAreaTask: leads agents to explore an established portion of the space.
- HealTask: for which a set of agents must rescue a defined animal (and must correspondingly have "rescue capabilities").

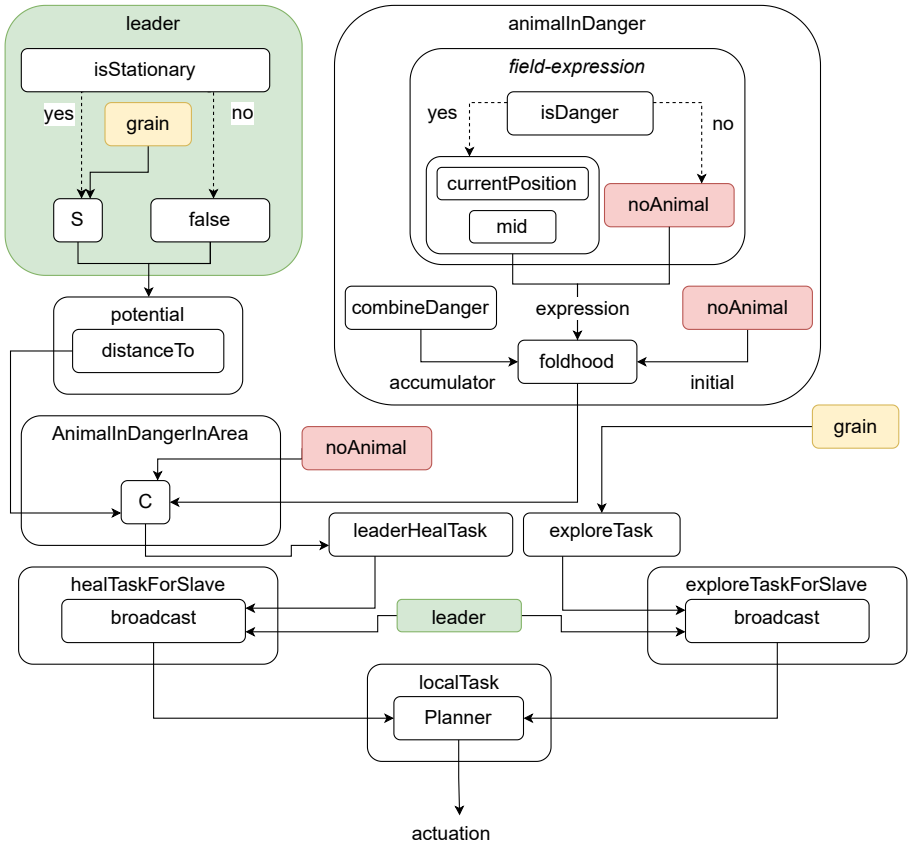
At runtime, the program identifies agent roles (healer, explorer, stationary) according to the local behaviour sensed. There are 80 mobile nodes (40 explorers and 40 healers), 20 stations and 100 animals. Only healers can rescue animals. The animals are divided into five independent groups. Each group moves according to the random waypoint logic [55]. An animal in danger needs a variable number of healers near to him to be rescued. The program follows the Self-organising Coordination Regions (SCR) pattern [44]:

1. leader election is made in the stationary nodes (via `S`);
2. mobile nodes sense animals in danger and send the local information to the leader (via `C`);
3. the leader chooses what is the animal that needs to be rescued;
4. the leader shares its choice (via `broadcast`);
5. the slaves act according to the leader choice.

To be more specific, the program first tries to detect nodes' role checking how much each node has moved in a defined time window:

```
val movementWindow = 6 // a domain-specific parameter
val movementThr = 1.0 //another domain-specific parameter
val trajectory = recentValues(movementWindow, currentPosition())
val lastPointInTrajectory = trajectory.reverse.head
val distanceApprox = trajectory.head.distanceTo(lastPointInTrajectory)
mux (distanceApprox <= movementThr) "stationary" else "explorer"
```

After, the collective chooses leaders using block `S` that is executed only where `isStationary` is `true` (using `branch`). At this point, mobile nodes check if there is an animal in danger in their neighborhood (i.e., they check the status of a special sensor called danger). To do this, we use the operator `foldhood(init)(acc)(expr)` to aggregate values over neighborhoods with operator `acc` and initial/null value `init`; the values that are aggregated are the results of the neighbors' evaluations of `expr` (where the neighbor-specific parts are defined through operator `nbr`). Then the information is sent to leaders through the `C` construct. This block aggregates and moves the data toward a potential field. In this case, the potential field is centered where the leader value is `true`. Leaders then receive a map specifying the approximate location of animals in danger and their ID. So they peek an animal with a local policy (e.g., choosing the nearest node) and then they create a HealTask that has the goal of directing healers to a chosen animal to rescue it. Besides, leaders share tasks that lead the explorers to stay in the leader's area of influence (i.e., creating a ExploreAreaTask). At this point, tasks are shared across the zones with the `broadcast` function. Finally, the slaves, who receive the various tasks created, choose which one to execute according to their role and local intentions (e.g., a node may not listen to the collective's decision and go their way). With ScaFi, this behavior can be synthesized as described in Figure 3b.



(**a**)

```
val leader = branch(isStationary) { S(grain) } { false } //(1.) leader election
//.. (2.) sense animals in danger ..
val noAnimal : Map[ID, P] = Map.empty
val animalsInDanger = {
  foldhood(noAnimal)(combineDangerMap){
    mux(nbr(isDanger)) {
      Map(nbr(mid() -> currentPosition()))
    } { noAnimal }
  }
}
val potential = distanceTo(leader)
//(2.) send information to the leader
val animalDangerInArea = C(potential, combineDangerMap, animalsInDanger, noAnimal)
//(3.) leader chooses an animal to rescue
val leaderHealTask = animalDangerInArea.toSeq.sortBy {
  case (_, p) => p.distance(currentPosition()) //choosing policy
}.headOption.map {
  case (id, p) => HealTask(mid(), id, p)
}
val exploreArea = ExploreTask(mid(), currentPosition(), grain) //for explorers
//(4.) and share its choice via broadcast
val healTaskForSlave =  broadcast(leader, leaderHealTask)
val exploreAreaToSlave = broadcast(leader, exploreArea)
//(5.) slave choose the task according its role, intentions and leader choice
val collectiveTasks = Seq(healTaskForSlave, exploreAreaToSlave)
val localTask = Planner.eval(collectiveTasks) //the task choice is encapsulated here..
val actuation = localTask.call(this) // produces data in order to achieve the task chosen
```

(**b**)

Figure 3. Wildlife monitoring and rescue example: architecture and program. (**a**) Diagram corresponding to the code below. This figure uses the same notation of Figure 2b. In addition, we use a same colour to denote multiple references to the same functional block. (**b**) The ScaFi code snippet for the wildlife program behaviour.

Next, we briefly link different behavior with different levels/types of autonomy. From the local perspective, each agent: *(i)* moves around the environment; *(ii)* executes tasks. Each behavior has an endogenous and exogenous aspect: with this program, we can specify how agents should behave (endogenous) but we have not total control over them because they maintain a certain level of exogenous autonomy. For example, when an agent receives the task to rescue an animal, he might decide to rescue another animal because he finds another one closer or because in that direction, there are obstacles. Moreover, the endogenous/exogenous autonomy level depends on the agent type. Indeed, we cannot control animal movements via the smart collar, so they maintain a total level of exogenous autonomy with respect to the program. From a collective perspective, the MAS performs: *(i)* multileader election, *(ii)* animal dangers sensing, and *(iii)* task selection to reduce the animal in danger in zones. With ScaFi the overall behaviour is intentionally described as:

```
val targets = animalsInDanger()
branch(isAnimal){doNothing} {rescue(targets)}
```

where `rescue(targets)` performs the code above and `animalsInDager` executes the neighbourhood perception danger sensor.

What we want to enforce in this experiment is that individual autonomy influences collective autonomy. For example, the leader election happens on stationary nodes. However, the node stationary rule depends on the local agent activity (mainly how they move), which is an exogenous behavior. So, what we expect is that most the system is locally autonomous less the collective autonomy influences the overall behavior.

4.2. Performance Metrics and Parameters

In simulation evaluation, we consider whether functional and non-functional aspects. The main functional metric is the rescue count, which describes how many animals are rescued during the simulation. Another condition that will be evaluated is the number of leaders without healers necessary to heal an animal. A non-functional value verified is the average distance to the leader. Ideally, agents might be arranged to cover a zone uniformly.

We also introduce a metric that represents the general performance of a simulation run. Given a period t expressed in seconds, we sample the experiment at each second. For each sample, we evaluate two parameters, *events* and *healed count*. Given a sample t, events measure how many animals turn on danger during the simulation until the t. Indeed, the healed count tells how many animals were rescued by the collective. Hypothetically, if the system is completely reactive, healed count and events should have the same value. Hence, the greater is the distance between these two values, the worse system perform. So, as comprehensive system performance, we decide to use Root Mean Squared Error (RMSE):

$$Error = \sqrt{\frac{1}{t} * \sum_{i=1}^{t} (events_i - healedcount_i)^2}$$

What we expect from these simulations is that selfish settings bring worse performance. Indeed, here we need a collective choice to rescue an animal. We test the application varying these parameters:

- p: is the probability to follow the collective choice, $1 - p$ is the probability to act selfishly; the bigger is p the lesser the agent is autonomous with respect to the collective goals.
- healer count: the nodes needed to rescue an animal. A higher value of healer count needs greater control on local agent behaviour in order to accomplish the collective task

4.3. Results and Discussion

We now present the key results produced by the different simulation runs we conducted. In Figure 4 there is a graphical result obtained in Alchemist. We had varied p by

four values (0, 0.25, 0.5, 0.75 and 1) and healer count by four values (2, 4, 6, 8). For each p and healer count combination, 50 simulation runs are performed. The data reported in Figure 5 contains the average behavior of those simulations. In Figure 6, the average error is reported for each combination of healer count and p. Each run has lasts 300 s. Furthermore, after 100 s, no other animals turn their status in danger. It helps us to see how what system is faster to return in a stable configuration.

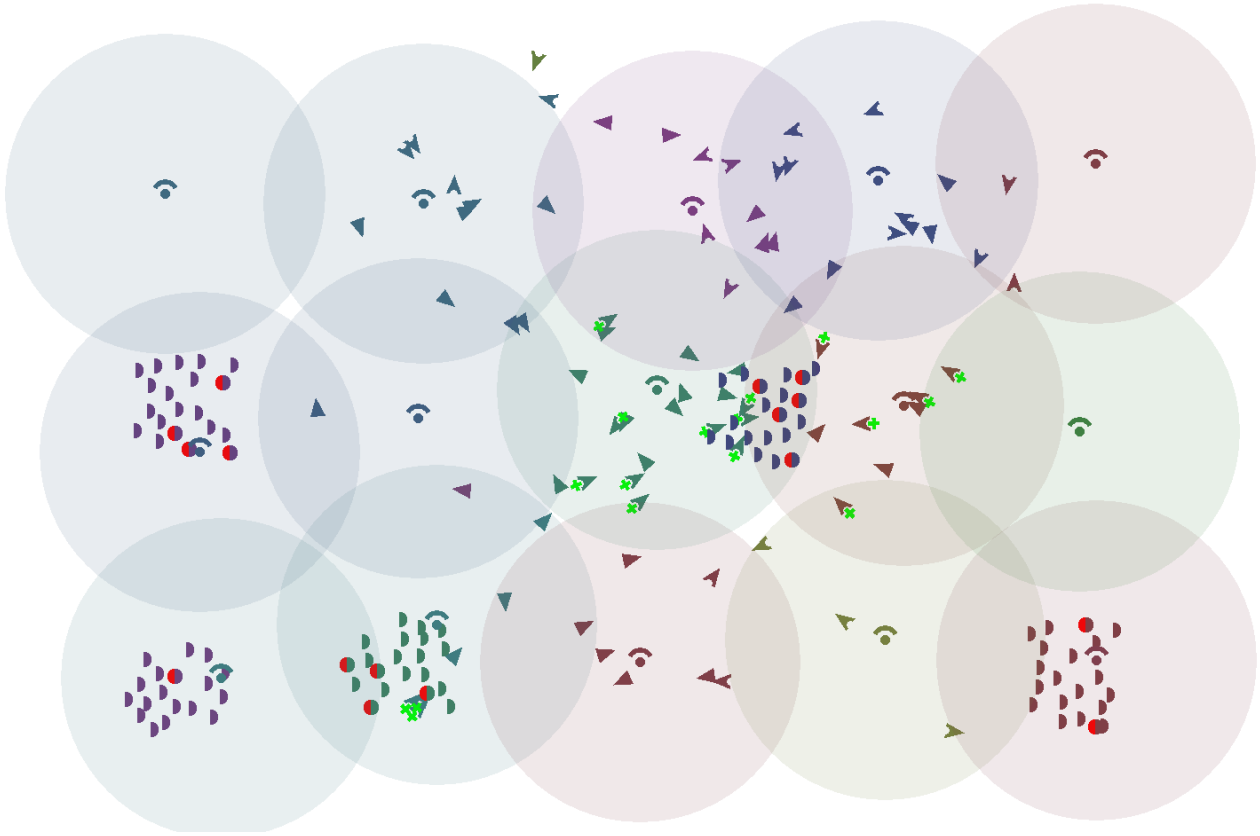


Figure 4. Wildlife monitoring simulation screenshot. The "wi-fi" like symbol represents a Station node. Each Station has an influence area (displayed with a colored circle). Mobile nodes are represented with a triangle. Healers have a half-moon on the shorter side. When healers have an animal target, a green cross appears. Animals are drawn as a half-circle. Each group has a uniform color. When they turn in danger, a half red circle comes out.

In general, the results confirm our thesis: the more agents act autonomously, the worse the system behaves. However, the performance depends upon the application domain. Indeed, when healer count is little, the overall behaviors aren't so different. Because even if agents do not collaborate, there is a high probability that two Mobile nodes are near to the same animal in danger. We see the benefits of collective choice with higher healer count values (Figure 6). For example, when healer count is 8 the system at $p = 1$ performs better than all other configurations. We want to emphasize how the difference w.r.t. the other p values grows as healer count increases. The difference instead becomes marked when healer count is 8, since the task has a greater need for collective choice and even small local selfish choices lead to worse performance. Outside of the functional aspects, we can clearly see that the higher the p value the more the system follows joint intentionality: agents arrange themselves equally in each area (having a higher average) and they cover the zones uniformly. This was an expected result, but it also makes us understand that even if the system loses some level of collectivity, it still manages to perform well (for example, if 25% of the time agents act independently, namely when p is 0.75, the overall performance is practically equal to when the agents always follow the collective choice). In general, however, we can observe that even if the system control is not total, the emerging

behavior is comparable with others. It is because all agents must adapt to the aggregate protocol, losing some of their autonomy.

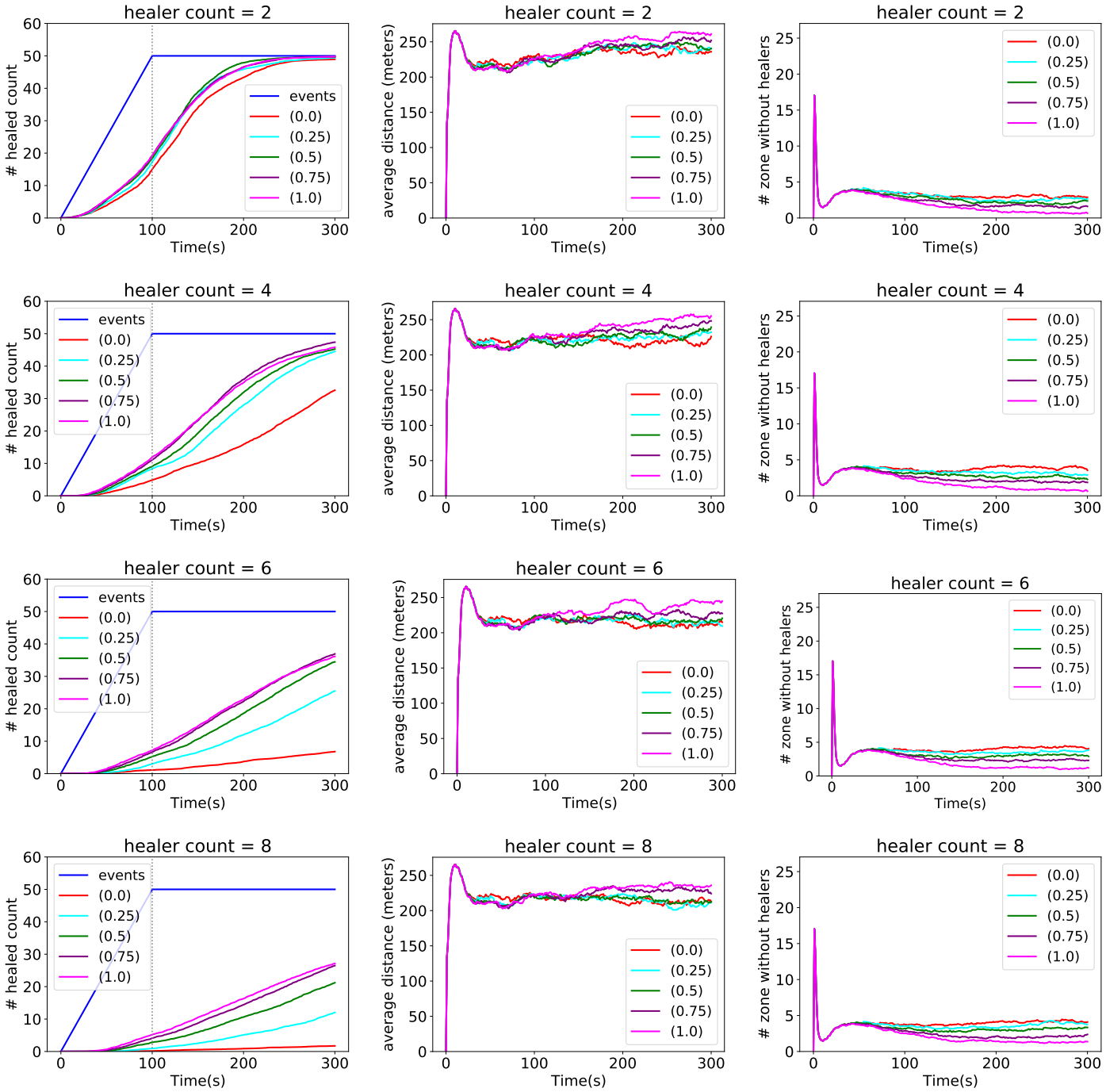


Figure 5. Experiment results. In each plot, the color identifies the p parameter. The first column shows how many animals are rescued during the simulation. The plots in the second column show the average distance from the leader. The plots in the last column point out how many zones have not enough healers to rescue an animal.

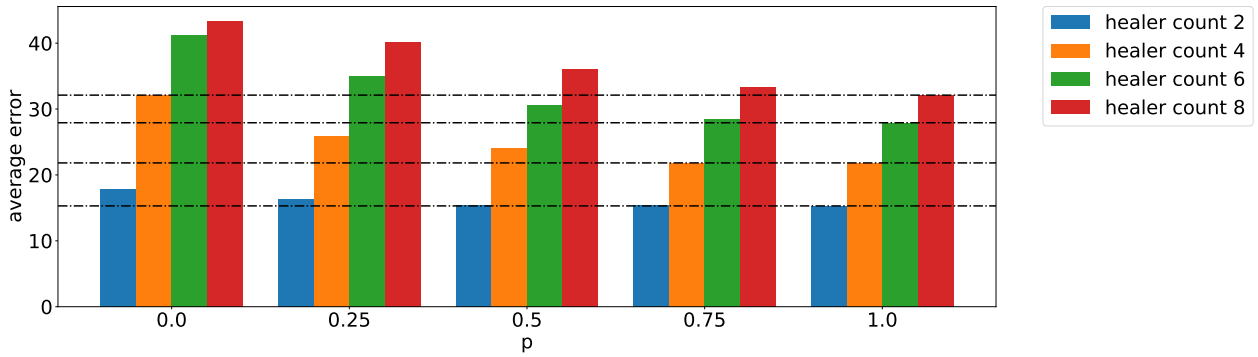


Figure 6. Average error (RMSE) for each combination of p and healer count. Horizontal lines mark the performance of p = 1.0.

4.4. Final Remarks

In this experiment, we deliberately did not consider the following aspects (even if they could affect the system performance), which are abstracted away: how the system detects animals in danger; the physical/hardware details of system nodes; how specific capabilities provided by the different node roles are implemented as concrete actions. Defining when an animal is in danger is a complex task per se and is strongly domain-dependent. In [56], the authors use UAVs to verify the presence of poachers. Concerning geofencing, in [57] the authors use a collar equipped with GPS to verify when animals escape from a boundary and mark them in danger. Animal immobility is another alarm signal used in [58]. Roles are introduced in this paper, though they are not present in other works (to the best of our knowledge), as they are functional to divide the work between nodes. In general, we expect that nodes could either be humans (cf., healers), UAVs (cf., explorers), or installed gateways. In the survey [59], some interesting work such as [60] does not require human intervention to rescue animals (cf., healer role). Finally, we want to underline that our approach is on a higher level of abstraction, focussing on coordination and execution of collective autonomous behavior. Environmental aspects are perceived through sensors that can be as simple (e.g., temperature sensor) or as complex as desired (e.g., a scheduler to choose tasks).

5. Research Roadmap

In the following, we identify a roadmap of two research directions to unveil the full potential of aggregate programming for collective autonomy: they comprise expressing collective autonomous behavior (Section 5.1) and achieving reliable collective autonomy (Section 5.2).

5.1. Expressing Collective Autonomous Behaviour

Addressing problems at a suitable level of abstraction is key in modeling and programming. In this section, we cover two interesting research directions related to specifying collective autonomy, and point out a few starting points in aggregate programming research.

5.1.1. Cognitive Collectives

Notions like distributed cognition and group minds are often leveraged in sociology and cognitive sciences to explain collective processes. In some works, such as in [61], a dynamical systems approach is used to model and understand the emergence of collective behavior and collective forms of consciousness. There, formal arguments are provided to explain how groups exert downward causation on their components. In the aggregate approach, groups or collectives are the target of programming, and the downward causation is a straightforward consequence of being part of the aggregate. However, collective cognitive states are not explicitly represented, but rather mixed in the aggregate programs, which can be seen as shared plans. Indeed, it would be interesting to investigate, along the

lines of [36], whether more explicit representations of collective cognition might help to drive group and individual behavior.

5.1.2. Collective Autonomy and Structural Organisation

One of the first works analyzing the relationship between organizational structure and autonomy is by Schillo [62]. Schillo proposes a framework based on the notion of delegation, draws a link between self-organization and adjustable autonomy, and defines a spectrum of seven organizational forms of increasing coupling between agents in a MAS: single-agent system, task delegation, virtual enterprise, cooperation, strategic network, group, cooperation. In [63], Horling et al. provide a survey of MAS organizational paradigms. They identify ten kinds of organizational structures: adhocracies, hierarchies, holarchies, coalitions, teams, congregations, societies, federations, markets, and matrix-like organizations. We believe that more work is needed for better understanding and specifying the inter-dependencies between structures and collective autonomous behavior.

For instance, approaches to MAS programming based on the notion of a team (i.e., a sub-collective) have been proposed in the past [64–66]. In the context of aggregate computing, the aggregate process abstraction—modeling a concurrent, dynamically scoped aggregate computation—has been recently introduced to extend the practical expressiveness of the paradigm. Aggregate processes, by defining concurrent activities with a dynamic and possibly overlapping scope, would support the specification of the autonomous behavior of multiple collectives. In this respect, it would be interesting to investigate the relationships between the autonomies of different collectives, as well as the interaction between the autonomies of a collective and its sub-collectives. Approaches inspired by holonic MASs [48], such as SARL [27], may prove useful or insights when considering multiple levels of autonomy and hierarchical organizations.

5.2. Reliable Collective Autonomy

Related to the ability of expressing collective autonomous behavior is the extent to which specifications lead to properties for dependable system operation that can be promoted, verified, and formally guaranteed.

5.2.1. Safety and Guarantees

Currently, few results are available for programming reliable collective adaptive behaviors. In the context of aggregate programming, major formal results include self-stabilisation [11] and eventual consistency to device distribution [39]. These results, however, are typically valid for restricted fragments of the field calculus. Moreover, self-stabilization does not say much about the speed of convergence, nor the ability of an algorithm to withstand continuous change. Therefore, typical validation approaches also include simulation [67] as a key step. Distributed runtime verification may also prove useful [68]. In general, more work is needed towards methods, both formal and lightweight, to verify the correctness and provide guarantees about global results in a certain range of environments and conditions.

5.2.2. Norms and Trust

Among the notions studied in literature to promote good cooperation between agents there are norms and trust. Through norms, and corresponding mechanisms for enforcing them (such as sanctions, rewards, and institutions), it is possible to regulate individual and then collective activity for social benefits. The problem is to make the MAS determine what deviant behavior is, detect it, and take corresponding countermeasures. Norms may be determined collectively, through processes of agreement or conflict resolution [69]. However, few results are available on conflicts among multiple norms and steering of their emergent effects at the collective level [69].

Trust can also be used as a way to reduce cooperation inefficiency and issues. An excess of individual autonomy may lead to deviance. In large-scale MASs, even few cases

of deviance may result in serious emergent effects at the collective level [70]. Preliminary work in aggregate programming research has been carried out to exclude non-trusted devices from the collective computation [71], leveraging a notion of trust field mapping any agent of the MAS to a trust score that is computed collectively. We believe further work is required on these topics to promote reliable (collective) autonomy.

5.3. Applications

Multi-agent systems and technologies span various application domains [72] including e-commerce [73], health care [74], logistics [75], robotics [76], manufacturing [77] and energy [78,79]: they are potential application areas for the approach to aggregate programming presented in this paper. The survey by Müller et al. [72] provides an overview of the impact of deployed MAS-based applications, showing that MAS technology has already been successful in various sectors. It is expected that MASs would be increasingly significant in the future, as more and more devices get deployed in our environments (cf. IoT, CPS, and related trends) and visions like autonomic computing continue to develop, fostering a pervasive embedding of computational autonomy at various levels. This is also plausible for collective autonomy, as applications involving (cyber–physical) collectives of (variously autonomous) actors seeking global goals emerge—cf. applications in smart city [80], swarm robotics [81], mobile social crowdsensing [82,83], and smart infrastructures contexts [51,84]. In particular, programmable approaches to collective autonomy could contribute to research and applications of computational collective intelligence [85], namely the field studying groups and their ability to implement effective decision-making, coordinated action, and cooperative problem-solving. However, we also remark that approaches to designing and programming MASs at the collective level should not be considered as omni-comprehensive approaches that deal with every aspect of a system; instead, they represent a tool that can be used to address certain problems (i.e., promoting global behavior and properties) at a suitable abstraction level. In this sense, a further challenge to be addressed in the future is the integration of collective-level approaches with individual-level approaches and traditional paradigms, promoting the vision of system development through integration of multiple perspectives and viewpoints [86].

6. Conclusions

In this paper, we consider the problem of programming the collective autonomous behavior of multi-agent systems. We consider the aggregate programming paradigm, a framework founded on a calculus of computational fields originally introduced to express the coordination and self-organisation logic of spatially situated systems. We analyze the support provided by aggregate computing by a MAS perspective, and, as a contribution, (i) interpret its execution protocol as an agent control architecture; and (ii) analyze aggregate programs by the point of view of individual and collective autonomy. Finally, we provide some simulation-based experiments, to show how the framework supports analysis of autonomy-related aspects, and discuss research gaps, pointing out opportunities for new research.

The various goals described in the introduction have been addressed as follows. The literature review in Section 2 provides a variegated view of autonomy in software engineering and MAS, with emphasis on actionable notions and programming. This helped us to position the contribution of Section 3 on aggregate computing, where multiple autonomy-related notions are supported (also enabling adjustable autonomy) but implicit, and had never been unveiled in previous publications. Explicit mechanisms and extensions for (adjustable) autonomy could be considered in the future. Moreover, the contribution of the view of the aggregate execution model as an agent control architecture represents a step towards comparison and integration with other architectures—which is left as interesting future work. The case study of Section 4 shows how the discussed framework enables functional specification of MASs exhibiting a form of collective autonomy as well as parameterized behaviors where individual and collective autonomy can be adjusted

off-line. Qualitatively, it shows that the approach is feasible and may scale with complexity. Self-organizing algorithms and patterns for on-line autonomy adjustment in aggregate systems as well as more quantitative analysis of trade-offs can be considered in future research. Finally, the discussion in Section 5 highlights significant research directions and application domains which complement the above discussion.

We argue that it is important to directly address the collective dimension of MASs, rather than programming individual agents and then verifying that local behaviors lead to the intended, but not explicitly captured, global behavior. It is a matter of abstraction and addressing concerns from a proper perspective. As discussed, specifications of collective autonomous behavior, such as aggregate programs written in ScaFi, provide natural support for adjustable autonomy, and this could pave the way to the integration with traditional agent control architectures. However, more work is needed to ensure that the collective specifications result in acceptable emergent behaviors, and hence in reliable forms of collective autonomy.

Author Contributions: Text.

Funding: This research received no external funding.

Institutional Review Board Statement: Text.

Informed Consent Statement: Text.

Data Availability Statement: Text.

Conflicts of Interest: Text.

References

1. Kephart, J.O.; Chess, D.M. The Vision of Autonomic Computing. *Computer* **2003**, *36*, 41–50.
2. de Lemos, R.; Giese, H.; Müller, H.A.; Shaw, M.; Andersson, J.; Litoiu, M.; Schmerl, B.R.; Tamura, G.; Villegas, N.M.; Vogel, T.; et al. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*; Springer: Berlin/Heidelberg, Germany, 2010; Volume 7475, pp. 1–32.
3. Wooldridge, M.J. *An Introduction to MultiAgent Systems*, 2nd ed.; Wiley: Hoboken, NJ, USA, 2009.
4. Ferber, J. *Multi-Agent Systems—An Introduction to Distributed Artificial Intelligence*; Addison-Wesley-Longman: Boston, MA, USA, 1999.
5. Weyns, D.; Omicini, A.; Odell, J. Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi Agent Syst.* **2007**, *14*, 5–30.
6. Beal, J.; Dulman, S.; Usbeck, K.; Viroli, M.; Correll, N. Organizing the Aggregate: Languages for Spatial Computing. *arXiv* **2012**, arXiv:1202.5509.
7. Viroli, M.; Beal, J.; Damiani, F.; Audrito, G.; Casadei, R.; Pianini, D. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.* **2019**, 100486, doi:10.1016/J.Jlamp.2019.100486.
8. Beal, J.; Pianini, D.; Viroli, M. Aggregate Programming for the Internet of Things. *Computer* **2015**, *48*, 22–30.
9. Audrito, G.; Viroli, M.; Damiani, F.; Pianini, D.; Beal, J. A Higher-Order Calculus of Computational Fields. *ACM Trans. Comput. Log.* **2019**, *20*, 5:1–5:55.
10. Beal, J.; Viroli, M. Building Blocks for Aggregate Programming of Self-Organising Applications. In Proceedings of the Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, UK, 8–12 September 2014; pp. 8–13, doi:10.1109/SASOW.2014.6.
11. Viroli, M.; Audrito, G.; Beal, J.; Damiani, F.; Pianini, D. Engineering Resilient Collective Adaptive Systems by Self-Stabilisation. *ACM Trans. Model. Comput. Simul.* **2018**, *28*, 16:1–16:28. doi:10.1145/3177774.
12. Malone, T.W.; Crowston, K. The interdisciplinary study of coordination. *ACM Comput. Surv.* **1994**, *26*, 87–119.
13. Odell, J. Objects and Agents Compared. *J. Object Technol.* **2002**, *1*, 41–53.
14. Franklin, S.; Graesser, A. *Is It an Agent, or just a Program? A Taxonomy for Autonomous Agents*; International Workshop on Agent Theories, Architectures, and Languages; Springer: Berlin/Heidelberg, Germany, 1996; pp. 21–35.
15. Castelfranchi, C.; Falcone, R. Founding Autonomy: The Dialectics Between (Social) Environment and Agent's Architecture and Powers. In *International Workshop on Computational Autonomy*; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2969, pp. 40–54.
16. Castelfranchi, C. Guarantees for autonomy in cognitive agent architecture. In *International Workshop on Agent Theories, Architectures, and Languages*; Springer: Berlin/Heidelberg, Germany, 1994; pp. 56–70.
17. De Silva, L.; Meneguzzi, F.; Logan, B. BDI Agent Architectures: A Survey. In Proceedings of the International Joint Conferences on Artificial Intelligence, Yokohama, Japan, 11–17 July 2020.

18. Sekiyama, K.; Fukuda, T. Dissipative structure network for collective autonomy: Spatial decomposition of robotic group. In *Distributed Autonomous Robotic Systems 2*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 221–232.
19. Bottazzi, E.; Catenacci, C.; Gangemi, A.; Lehmann, J. From collective intentionality to intentional collectives: An ontological perspective. *Cogn. Syst. Res.* **2006**, *7*, 192–208.
20. Ray, P.; O'Rourke, M.; Edwards, D. Using collective intentionality to model fleets of autonomous underwater vehicles. In Proceedings of the OCEANS 2009, Biloxi, MS, USA, 26–29 October 2009; pp. 1–7.
21. Conte, R.; Turrini, P. Argyll-Feet giants: A cognitive analysis of collective autonomy. *Cogn. Syst. Res.* **2006**, *7*, 209–219.
22. Liu, J.; Jin, X.; Tsui, K.C. Autonomy-oriented computing (AOC): formulating computational systems with autonomous components. *IEEE Trans. Syst. Man Cybern. Part A* **2005**, *35*, 879–902.
23. Mao, X.; Wang, Q.; Yang, S. A survey of agent-oriented programming from software engineering perspective. *Web Intell.* **2017**, *15*, 143–163. doi:10.3233/WEB-170357.
24. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*; John Wiley & Sons: Hoboken, NJ, USA, 2007; Volume 8.
25. Finin, T.W.; Fritzson, R.; McKay, D.P.; McEntire, R. KQML As An Agent Communication Language. In Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, MD, USA, 29 November–2 December 1994; ACM: 1994; pp. 456–463, doi:10.1145/191246.191322.
26. Ricci, A.; Piunti, M.; Viroli, M.; Omicini, A. Environment Programming in CArtAgO. In *Multi-Agent Programming, Languages, Tools and Applications*; Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 259–288, doi:10.1007/978-0-387-89299-3_8.
27. Rodriguez, S.; Gaud, N.; Galland, S. SARL: A General-Purpose Agent-Oriented Programming Language. In Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Warsaw, Poland, 11–14 August 2014; pp. 103–110, doi:10.1109/WI-IAT.2014.156.
28. Hübner, J.F.; Sichman, J.S.; Boissier, O. Developing organised multiagent systems using the MOISE$^+$ model: Programming issues at the system and agent levels. *Int. J. Agent Oriented Softw. Eng.* **2007**, *1*, 370–395. doi:10.1504/IJAOSE.2007.016266.
29. Cardoso, R.C.; Ferrando, A. A Review of Agent-Based Programming for Multi-Agent Systems. *Computers* **2021**, *10*, 16. doi:10.3390/computers10020016.
30. DeHon, A.; Giavitto, J.; Gruau, F. (Eds.) *Computing Media and Languages for Space-Oriented Computation*; Schloss Dagstuhl-Leibniz-Zentrum für Informatik: Schloss Dagstuhl, Germany, 2007; Volume 06361.
31. Mamei, M.; Zambonelli, F. *Field-Based Coordination for Pervasive Multiagent Systems*; Springer Series on Agent Technology; Springer: Berlin/Heidelberg, Germany, 2006. doi:10.1007/3-540-27969-5.
32. Parunak, H.V.D.; Brueckner, S.; Sauter, J.A. Digital pheromone mechanisms for coordination of unmanned vehicles. In Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, Bologna, Italy, 15–19 July 2002; pp. 449–450, doi:10.1145/544741.544843.
33. Casadei, R.; Viroli, M.; Audrito, G.; Damiani, F. FScaFi: A Core Calculus for Collective Adaptive Systems Programming. Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles. In Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, 20–30 October 2020; Lecture Notes in Computer Science; Margaria, T., Steffen, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2020; Volume 12477, pp. 344–360, doi:10.1007/978-3-030-61470-6_21.
34. Casadei, R.; Viroli, M.; Audrito, G.; Pianini, D.; Damiani, F. Engineering collective intelligence at the edge with aggregate processes. *Eng. Appl. Artif. Intell.* **2021**, *97*, 104081. doi:10.1016/j.engappai.2020.104081.
35. Casadei, R.; Pianini, D.; Viroli, M. Simulating large-scale aggregate MASs with Alchemist and Scala. In Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), Gdansk, Poland, 11–14 September 2016, pp. 1495–1504.
36. Viroli, M.; Pianini, D.; Ricci, A.; Croatti, A. Aggregate plans for multiagent systems. *Int. J. Agent Oriented Softw. Eng.* **2017**, *5*, 336–365.
37. Casadei, R.; Pianini, D.; Placuzzi, A.; Viroli, M.; Weyns, D. Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment. *Future Internet* **2020**, *12*, 203. doi:10.3390/fi12110203.
38. Pianini, D.; Casadei, R.; Viroli, M.; Mariani, S.; Zambonelli, F. Time-Fluid Field-Based Coordination through Programmable Distributed Schedulers. *arXiv* **2020**, arXiv:2012.13806.
39. Beal, J.; Viroli, M.; Pianini, D.; Damiani, F. Self-Adaptation to Device Distribution in the Internet of Things. *ACM Trans. Auton. Adapt. Syst.* **2017**, *12*, 12:1–12:29. doi:10.1145/3105758.
40. Hollander, C.D.; Wu, A.S. The Current State of Normative Agent-Based Systems. *J. Artif. Soc. Soc. Simul.* **2011**, *14*. doi:10.18564/jasss.1750.
41. Mostafa, S.A.; Ahmad, M.S.; Mustapha, A. Adjustable autonomy: A systematic literature review. *Artif. Intell. Rev.* **2019**, *51*, 149–186. doi:10.1007/s10462-017-9560-8.
42. Dennett, D.C. *The Intentional Stance*; MIT Press: Cambridge, MA, USA, 1989.
43. Huebner, B. *Macrocognition: A Theory of Distributed Minds and Collective Intentionality*; Oxford University Press: Oxford, UK, 2014.
44. Casadei, R.; Pianini, D.; Viroli, M.; Natali, A. Self-organising Coordination Regions: A Pattern for Edge Computing. In Proceedings of the International Conference on Coordination Languages and Models, 2019; Springer: Cham, Switzerland, 2019; pp. 182–199.

45. Mo, Y.; Beal, J.; Dasgupta, S. An Aggregate Computing Approach to Self-Stabilizing Leader Election. In Proceedings of the 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, 3–7 September 2018; pp. 112–117, doi:10.1109/FAS-W.2018.00034.
46. Audrito, G.; Casadei, R.; Damiani, F.; Viroli, M. Compositional Blocks for Optimal Self-Healing Gradients. In Proceedings of the 2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Tucson, AZ, USA, 18–22 September 2017.
47. Wolf, T.D.; Holvoet, T. Designing Self-Organising Emergent Systems based on Information Flows and Feedback-loops. In Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, 9–11 July 2007; pp. 295–298, doi:10.1109/SASO.2007.16.
48. Rodriguez, S.; Hilaire, V.; Gaud, N.; Galland, S.; Koukam, A. Holonic Multi-Agent Systems. In *Self-organising Software—From Natural to Artificial Adaptation*; Serugendo, G.D.M., Gleizes, M., Karageorgos, A., Eds.; Natural Computing Series; Springer: Berlin/Heidelberg, Germany, 2011; pp. 251–279, doi:10.1007/978-3-642-17348-6_11.
49. Nicola, R.D.; Loreti, M.; Pugliese, R.; Tiezzi, F. A Formal Approach to Autonomic Systems Programming: The SCEL Language. *ACM Trans. Auton. Adapt. Syst.* **2014**, *9*, 7:1–7:29. doi:10.1145/2619998.
50. Pianini, D.; Montagna, S.; Viroli, M. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation* **2013**, *7*, 202–215. doi:10.1057/jos.2012.27.
51. Casadei, R.; Viroli, M. Coordinating Computation at the Edge: A Decentralized, Self-Organizing, Spatial Approach. In Proceedings of the 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC), Rome, Italy, 10–13 June 2019; pp. 60–67. doi:10.1109/FMEC.2019.8795355.
52. Gonzalez, L.F.; Montes, G.A.; Puig, E.; Johnson, S.; Mengersen, K.L.; Gaston, K.J. Unmanned Aerial Vehicles (UAVs) and Artificial Intelligence Revolutionizing Wildlife Monitoring and Conservation. *Sensors* **2016**, *16*, 97. doi:10.3390/s16010097.
53. Ayele, E.D.; Meratnia, N.; Havinga, P.J.M. Towards a New Opportunistic IoT Network Architecture for Wildlife Monitoring System. In Proceedings of the 9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, 26–28 February 2018; pp. 1–5. doi:10.1109/NTMS.2018.8328721.
54. Fatima, S.S.; Wooldridge, M.J. Adaptive task resources allocation in multi-agent systems. In Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS 2001, Montreal, QC, Canada, 28 May–1 June 2001; André, E., Sen, S., Frasson, C., Müller, J.P., Eds.; ACM: 2001; pp. 537–544. doi:10.1145/375735.376439.
55. Mao, S. Chapter 8 - Fundamentals of communication networks. In *Cognitive Radio Communications and Networks*; Wyglinski, A.M., Nekovee, M., Hou, Y.T., Eds.; Academic Press: Oxford, UK, 2010; pp. 201–234. doi:10.1016/B978-0-12-374715-0.00008-3.
56. Mulero-Pázmány, M.; Stolper, R.; van Essen, L.D.; Negro, J.J.; Sassen, T. Remotely Piloted Aircraft Systems as a Rhinoceros Anti-Poaching Tool in Africa. *PLoS ONE* **2014**, *9*, e83873. doi:10.1371/journal.pone.0083873.
57. Marini, D.; Llewellyn, R.; Belson, S.; Lee, C. Controlling Within-Field Sheep Movement Using Virtual Fencing. *Animals* **2018**, *8*, 31. doi:10.3390/ani8030031.
58. Wall, J.; Wittemyer, G.; Klinkenberg, B.; Douglas-Hamilton, I. Novel opportunities for wildlife conservation and research with real-time monitoring. *Ecol. Appl.* **2014**, *24*, 593–601. doi:10.1890/13-1971.1.
59. López, J.J.; Mulero-Pázmány, M. Drones for Conservation in Protected Areas: Present and Future. *Drones* **2019**, *3*, 10. doi:10.3390/drones3010010.
60. Hahn, N.; Mwakatobe, A.; Konuche, J.; de Souza, N.; Keyyu, J.; Goss, M.; Chang'a, A.; Palminteri, S.; Dinerstein, E.; Olson, D. Unmanned aerial vehicles mitigate human–elephant conflict on the borders of Tanzanian Parks: A case study. *Oryx* **2016**, *51*, 513–516. doi:10.1017/s0030605316000946.
61. Palermos, S.O. The Dynamics of Group Cognition. *Minds Mach.* **2016**, *26*, 409–440. doi:10.1007/s11023-016-9402-5.
62. Schillo, M. Self-organization and adjustable autonomy: Two sides of the same coin? *Connect. Sci.* **2002**, *14*, 345–359. doi:10.1080/0954009021000068718.
63. Horling, B.; Lesser, V.R. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.* **2004**, *19*, 281–316. doi:10.1017/S0269888905000317.
64. Pynadath, D.V.; Tambe, M.; Chauvat, N.; Cavedon, L. Toward Team-Oriented Programming. In Proceedings of the Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL '99, Orlando, FL, USA,15–17 July 1999; Jennings, N.R., Lespérance, Y., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1757, pp. 233–247. doi:10.1007/10719619_17.
65. Jarvis, J.; Rönnquist, R.; McFarlane, D.C.; Jain, L.C. A team-based holonic approach to robotic assembly cell control. *J. Netw. Comput. Appl.* **2006**, *29*, 160–176. doi:10.1016/j.jnca.2004.10.001.
66. Koutsoubelias, M.; Lalis, S. TeCoLa: A Programming Framework for Dynamic and Heterogeneous Robotic Teams. In Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, MobiQuitous 2016, Hiroshima, Japan, 28 November–1 December 2016; Hara, T., Shigeno, H., Eds.; ACM: 2016, pp. 115–124. doi:10.1145/2994374.2994397.
67. Mittal, S.; Risco-Martin, J.L. Simulation-based complex adaptive systems. In *Guide to Simulation-Based Disciplines*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 127–150.
68. Audrito, G.; Casadei, R.; Damiani, F.; Stolz, V.; Viroli, M. Adaptive distributed monitors of spatial properties for cyber-physical systems. *J. Syst. Softw.* **2021**, *175*, 110908.

69. dos Santos, J.S.; de Oliveira Zahn, J.; Silvestre, E.A.; da Silva, V.T.; Vasconcelos, W.W. Detection and resolution of normative conflicts in multi-agent systems: a literature survey. *Auton. Agents Multi Agent Syst.* **2017**, *31*, 1236–1282. doi:10.1007/s10458-017-9362-z.
70. Aldini, A. Design and Verification of Trusted Collective Adaptive Systems. *ACM Trans. Model. Comput. Simul.* **2018**, *28*, 9:1–9:27. doi:10.1145/3155337.
71. Casadei, R.; Aldini, A.; Viroli, M. Towards attack-resistant Aggregate Computing using trust mechanisms. *Sci. Comput. Program.* **2018**, *167*, 114–137.
72. Müller, J.P.; Fischer, K. Application Impact of Multi-agent Systems and Technologies: A Survey. In *Agent-Oriented Software Engineering—Reflections on Architectures, Methodologies, Languages, and Frameworks*; Shehory, O., Sturm, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; pp. 27–53. doi:10.1007/978-3-642-54432-3_3.
73. Fasli, M. *Agent Technology for e-Commerce*; Wiley: Hoboken, NJ, USA, 2007.
74. Bergenti, F.; Poggi, A. Multi-agent systems for e-health: Recent projects and initiatives. In Proceedings of the 10th International Workshop on Objects and Agents, 2009.
75. Burmeister, B.; Haddadi, A.; Matylis, G. Application of multi-agent systems in traffic and transportation. *IEE Proc. Softw. Eng.* **1997**, *144*, 51–60. doi:10.1049/ip-sen:19971023.
76. Dudek, G.; Jenkin, M.R.M.; Milios, E.E.; Wilkes, D. A taxonomy for multi-agent robotics. *Auton. Robots* **1996**, *3*, 375–397. doi:10.1007/BF00240651.
77. Lee, J.H.; Kim, C.O. Multi-agent systems applications in manufacturing systems and supply chain management: A review paper. *Int. J. Prod. Res.* **2008**, *46*, 233–265.
78. González-Briones, A.; De La Prieta, F.; Mohamad, M.S.; Omatu, S.; Corchado, J.M. Multi-agent systems applications in energy optimization problems: A state-of-the-art review. *Energies* **2018**, *11*, 1928.
79. Merabet, G.H.; Essaaidi, M.; Talei, H.; Abid, M.R.; Khalil, N.; Madkour, M.; Benhaddou, D. Applications of Multi-Agent Systems in Smart Grids: A survey. In Proceedings of the 4th International Conference on Multimedia Computing and Systems, ICMCS 2014, Marrakech, Morocco, 14–16 April 2014; pp. 1088–1094. doi:10.1109/ICMCS.2014.6911384.
80. Zanella, A.; Bui, N.; Castellani, A.P.; Vangelista, L.; Zorzi, M. Internet of Things for Smart Cities. *IEEE Internet Things J.* **2014**, *1*, 22–32. doi:10.1109/JIOT.2014.2306328.
81. Brambilla, M.; Ferrante, E.; Birattari, M.; Dorigo, M. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intell.* **2013**, *7*, 1–41. doi:10.1007/s11721-012-0075-2.
82. Ganti, R.K.; Ye, F.; Lei, H. Mobile crowdsensing: Current state and future challenges. *IEEE Commun. Mag.* **2011**, *49*, 32–39. doi:10.1109/MCOM.2011.6069707.
83. Bucchiarone, A.; D'Angelo, M.; Pianini, D.; Cabri, G.; De Sanctis, M.; Viroli, M.; Casadei, R.; Dobson, S. On the Social Implications of Collective Adaptive Systems. *IEEE Technol. Soc. Mag.* **2020**, *39*, 36–46. doi:10.1109/MTS.2020.3012324.
84. Annaswamy, A.M.; Malekpour, A.R.; Baros, S. Emerging research topics in control for smart infrastructures. *Annu. Rev. Control* **2016**, *42*, 259–270. doi:10.1016/j.arcontrol.2016.10.001.
85. Szuba, T. *Computational Collective Intelligence*; Wiley Series on Parallel and Distributed Computing; Wiley: Hoboken, NJ, USA, 2001.
86. Finkelstein, A.; Kramer, J.; Nuseibeh, B.; Finkelstein, L.; Goedicke, M. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *Int. J. Softw. Eng. Knowl. Eng.* **1992**, *2*, 31–57. doi:10.1142/S0218194092000038.