MacroSwarm: A Field-Based Compositional Framework for Swarm Programming

MacroSwarm: a Field-based Compositional Framework for Swarm Programming

Gianluca Aguzzi[0000−0002−1553−4561], Roberto Casadei[0000−0001−9149−949X], and Mirko Viroli[0000−0003−2702−5702]

Alma Mater Studiorum – Università di Bologna
{gianluca.aguzzi,roby.casadei,mirko.viroli}@unibo.it

Abstract. Swarm behaviour engineering is an area of research that seeks to investigate methods for coordinating computation and action within groups of simple agents to achieve complex global goals like collective movement, clustering, and distributed sensing. Despite recent progress in the study and engineering of swarms (of drones, robots, vehicles), there is still need for general design and implementation methods that can be used to define complex swarm coordination in a principled way. To face this need, this paper proposes a new field-based coordination approach, called MacroSwarm, to design fully composable and reusable blocks of swarm behaviour. Based on the macroprogramming approach of aggregate computing, it roots on the idea of modelling each block of swarm behaviour by a purely functional transformation of sensing fields into actuation description fields, typically including movement vectors. We showcase the potential of MacroSwarm as a framework for collective intelligence by simulation, in a variety of scenarios including flocking, morphogenesis, and collective decision-making.

Keywords: Swarm Behaviours · Field-based Coordination · Aggregate Computing · Collective Intelligence · Distributed Computing · DSLs.

1 Introduction

Recent technological advances foster a vision of *swarms* of mobile cyber-physical agents able to compute, coordinate with neighbours, and interact with the environment according to increasingly complex patterns, plans, and goals. Notable examples include swarms of drones and robots [44], fleets of vehicles [45], and crowds of wearable-augmented people [24]. In these domains, a prominent research problem is how to effectively engineer *swarm behaviour* [13], i.e., how to promote the emergence of desired global-level outcomes with inherent robustness and resiliency to changes and faults in the swarm or the environment. Complex patterns can emerge through the interaction of simple agents [12], and centralised approaches can suffer from scalability and dependability issues: as such, we seek for an approach based on suitable distributed coordination models and languages to steer the micro-level activity of a possibly large set of agents. This direction has been explored by various research threads related to

coordination like *macroprogramming* [16,35], *spatial computing* [10], *ensemble languages* [21,1], *field-based coordination* [31,33], and *aggregate computing* [50].

Though a number of approaches and languages have been proposed for specifying or programming swarm behaviour [4,15,22,28,29,30,34,41,51], a key feature that is generally missing or provided only to a limited extent is *compositionality*, namely the ability of combining blocks of simple swarm behaviour to construct swarm systems of increasing complexity in a controlled/engineered way. Additionally, most of existing approaches tend to be pragmatic, not formally-founded and quite ad-hoc: they enable construction of certain types of swarm applications but with limited support for analysis and principled design of complex applications (e.g. [30,22,41,15]). Exceptions that provide a formal approach exist but they are typically overly abstract, requiring additional effort to actually code and execute swarm control programs [32].

The goal of this work is to introduce a formally-grounded Application Program Interface (API), expressive and practical enough to concisely and elegantly encode a wide array of swarm behaviours. This is based on the field-based coordination paradigm [50] and the field calculus [9]: each block of swarm behaviour is captured by a purely functional transformation of sensing fields into actuation fields including movement vectors, and such a transformation declaratively captures the state/computation/interaction mechanisms necessary to achieve that behaviour. Practically, such specifications can be programmed as Scala scripts in the ScaFi framework [18,8], a reference implementation for field-based coordination and aggregate computing. Accordingly, we present MacroSwarm, a ScaFi-based framework to help programming with swarm behaviours by providing a set of blocks covering key swarming patterns as identified in literature [13]: flocking, leader-follower behaviours, morphogenesis, and team formation. To evaluate MacroSwarm, we show a use case that leverage our API in a simulated environment based on the Alchemist multi-agent system simulator [40].

The remainder of this paper is organised as follows. Section 2 provides context and motivation. Section 3 reviews background on aggregate computing. Section 4 presents the main contribution of the paper, MacroSwarm. Section 5 provides a simulation-based evaluation of the approach. Section 6 reviews related works on swarm programming. Finally, Section 7 provides a conclusion and future work.

2 Context and Motivation

Engineering the collective behaviour of swarms is a significant research challenge [13]. Two main kinds of design methods can be identified [13]: *automatic* design methods like evolutionary robotics [46] or multi-agent reinforcement learning [14], also called *behaviour-based* design, involving manually-implemented algorithms expressed via general-purpose or domain-specific languages (DSLs). Our focus is on the latter category of methods and especially on DSLs for expressing swarm behaviour (which are reviewed in Section 6).

Another main distinction is between *centralised* (*orchestration-based*) and *decentralised* (*choreographical*) approaches. In the former category, programs

generally specify tasks and relationships between tasks, and these descriptions are used by a centralised entity to command the behaviour of the individual entities of the swarm. By contrast, decentralised approaches do not rely on any centralised entity: each robot is driven by a control program and the resulting execution is decentralised (e.g., based on interaction with neighbours, like in Meld [4]). In this work, we focus on *decentralised* solutions, for they support resilience and scalability by avoiding single-points-of-failure and bottlenecks.

In the general context of behaviour-based swarm design, researchers have pointed out various issues [13,23] like a general lack of *top-down* design methods of collective behaviours (cf. the scientific issue of "emergence programming" [47] and "self-organisation steering" [25]), the problem of formal verification and validation [32], heterogeneity, and operational/maintenance issues (e.g., scalability, adaptation, and security).

To address top-down swarm programming, an approach should provide the means to define and compose blocks of high-level swarm behaviours.Regarding the kinds of blocks that can be provided, it is helpful to look at proposed taxonomies of collective/swarm behaviour. In a prominent survey on swarm engineering [13], collective behaviours are classified into (i) spatially-organising behaviours (e.g., pattern formation, morphogenesis), (ii) navigation behaviours (e.g., collective exploration, transport, and coordinated motion), (iii) collective decision-making (e.g., consensus achievement and task allocation), and (iv) others (e.g., human-swarm interaction and group size regulation).

Finally, we observe in the literature a rather sharp distinction between approaches leveraging formal methods for specifying swarm behaviour [32], also enabling verification, and more pragmatic approaches offering concrete DSLs that are more usable. In a recent survey on formal specification methods for swarm robotics [32] it is reported that a major limitation lies in (i) the tooling and (i) the formalisation of the "last step" of passing from a formal model to program code. Hence, we seek seek here for an approach that combines the benefits of formal methods and the pragmatism of concrete DSLs.

In summary, this work is motivated by the need of an approach for *formal-yet-practical top-down behaviour-based design of decentralised swarm behaviour*.

3 Background: Aggregate Computing

Aggregate computing [50] is a field-based coordination [33] and macroprogramming [16] approach especially suitable to express the collective adaptive [36] and self-organising behaviour of large groups of situated agents.

System model. In aggregate computing, a system can be simply modelled as a logical set of computing *nodes* (also called *devices*), where each node is equipped with *sensors* and *actuators*, and is connected with other nodes according to some *neighbouring relationships*. This abstract logical model does not prescribe particular technological solutions; instead, it uses minimal assumptions on the

capabilities of devices (e.g., regarding synchrony, connectivity, and computing power).

Execution model. The approach is generally used to program long-running control tasks that need several sensing, communication, computation, and actuation steps to be carried out. Accordingly, the execution model is based on (or can be understood as) a repeated execution, by each device, of asynchronous sense–compute–interact *rounds*—fundamentally mimicking self-organisation in biological systems [12]. For simplicity, we can consider each round to atomically consist of three steps:

- **Sense** – the node's *local context* is assessed, by sampling sensors and gathering the most recent (and not expired) message from any neighbour;
- **Compute** – the so-called *aggregate program* is evaluated against the local context, producing an output (which can be used to describe actuations) and an internal output (invisible to programmers), called an *export*, that contains the message to be sent to neighbours for coordination purposes;
- **Interact** – the export is sent to neighbours (logically, as a broadcast), and potential actuations can be performed.

In general, programs define the logic for spreading information from neighbourhood to neighbourhood, and progressively compute results eventually reaching convergence once no more changes perturb the system. Interestingly, device failure, message loss, and the like are automatically tolerated as assessed by context updates at the beginning of rounds. In order to understand how an aggregate program executed in this fashion promotes collective adaptive behaviour, we briefly present the programming model.

Programming model. Aggregate computing is based on the *(computational) field* abstraction [33]. A field is basically a function or map from devices to computational values. For instance, having a collection of devices query their temperature sensor would yield a field of real numbers denoting temperature readings, whereas a field of speed vectors could be use to denote the desired actuations to make a swarm move.

The *field calculus* [9] is the minimal core language at the basis of aggregate computing, which defines the primitives for expressing "space-time universal" [5] distributed computations in terms of field manipulations. Then, concrete languages like the Scala-internal DSL ScaFi (SCAla FIelds) [18,8] can be used to actually develop aggregate programs.

The reader can refer to [8] for a full presentation of programming with ScaFi. Here, we briefly introduce the main language constructs.

Construct `rep`: stateful field evolution. Consider the following example.

```
// def rep[T](init: T)(f: T => T): T
rep(0)(x => x+1) // type T=Int inferred
```

This purely local computation, when considered executed by all the devices in the system, yields a field of integers denoting the number of rounds executed by each device. This is obtained by applying function `f` to the value computed the previous round (or `init`, initially).

Construct `foldhood`/`nbr`: interaction with neighbours. Consider:

```
// def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
// def nbr[A](expr: => A): A
foldhood[Set[ID]](Set.empty)(_++_){ Set(nbr(mid())) }
```

It yields, in each device, the set of identifiers of all its neighbours. This is achieved by a purely functional fold over the collection of the singleton sets of neighbour identifiers, starting from the empty set, and aggregating using the set union operator (`++`). `mid()` provides the local identifier. Within the `foldhood`, a `nbr(e)` expression has the twofold role of sending and gathering the local value of `e` to/from neighbours. Note that constructs `rep` and `foldhood`/`nbr` can be combined to support the diffusion of information beyond direct neighbours.

Functional abstraction. New blocks can be defined with standard Scala functions:

```
def neighbouringField[T](f: => T): Set[T] =
  foldhood[Set[T]](Set.empty)(_++_){ Set(nbr(f)) }
def neighbourIDs(): Set[ID] = neighbouringField{ mid() }
```

Construct `branch`: splitting computation domains. Consider:

```
// def branch[A](cond: => Boolean)(th: => A)(el: => A)
branch(sense[Boolean]("hasTemperatureSensor")){
  val nearbyTemperatures: Set[Double] =
    neighbouringField{ sense[Double]("temperature") }
  // ...
}{ noOp }
```

Here, computation is split into separate subsets of devices. Notice that neighbourhoods are restricted in each computation branch. So, in the first branch, it is ensured that only the neighbours with a temperature sensor are folded over.

General resilient operators. It is possible to identify some general higher-level operators to account for common self-organisation patterns [49]. These will be leveraged in Section 4 and hence are briefly described.

- ***S**parse choice (leader election) [38].* Block `S(grain:Double):Boolean` can be used to yield a self-stabilising Boolean field which is `true` in a sparse set of devices located at a mean distance `grain`.
- ***G**radient-cast (distributed propagation) [49].* Block `G[T](source:Boolean,value:T,acc:T=>T):T` is used to propagate `value` from `source` devices outwards along the gradient [49] of increasing distances from them, transforming the value through `acc` along the way.

– ***C**ollect-cast (distributed collection) [6].* Block `C[T](sink:Boolean,value:T,acc(T,T)=>T):T` is used to summarise distributed information into `sink` devices, the `value`s provided by devices around the system, while aggregating information through `acc` along the gradient directed towards the sinks.

Examples further showing the compositionality of the approach are in Section 4.

Aggregate computing for swarm programming. To motivate why aggregate computing appears to be a good match for swarm programming, we briefly explain how it helps to address the challenges identified, as studied in previous papers (presented in Section 2).

– *Top-down behaviour-based design.* It is promoted by the *compositionality* and *collective stance* of aggregate computing, supported by the functional paradigm and the field abstraction [9,7].
– *Scalability.* Since execution is fully decentralised and asynchronous, the approach is scalable to hundreds, thousands, and even more devices [19].
– *Formal approach.* Aggregate computing and ScaFi are based on the field calculus [8,9], which enables formal analysis of programs and proofs of interesting properties like self-stabilisation [49], universality [5], and others [50].
– *Pragmatism.* Promoted by layers of abstractions, this is witnessed by open-source, maintained, concrete software artefacts like the ScaFi DSL [18], simulation platforms like Alchemist [40] and ScaFi-Web [2][1], and the possibility to devise libraries of high-level functions [19].
– *Operational flexibility.* Concrete aggregate computing systems can be deployed and operated using different architectural styles [17] and execution policies [39], supporting different technological and resource requirements.

4 MacroSwarm

This section presents the MacroSwarm approach and API. In particular, we describe its overall architecture, and the main blocks exposed by the API (summarised in Figure 1), which support the specification of a wide range of high-level swarm behaviours. The key idea in the design of MacroSwarm lies in the representation of a swarm behavioural unit as a function mapping sensing and parameter fields to actuation fields (often, velocity vectors). We have organised the API into multiple *modules*, capturing logically related sets of behaviours, and comprising more fundamental and reusable sets of behaviours as well as more application-specific sets (e.g., related to movement or team formation).

Movement blocks. These blocks control the movement of individual agents within the swarm. The simplest movement expressible with MacroSwarm is a collective constant movement (Figure 2a), described through a tuple like `Vector(x,y,z)` that devises the velocity vector of the swarm:

[1] `https://scafi.github.io/web/`

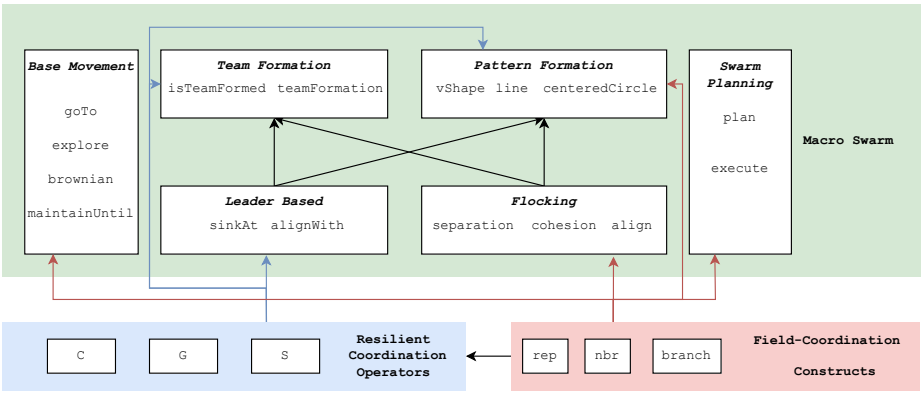


Fig. 1: MacroSwarm: architecture overview. The black boxes contained in the green rectangle represent the main modules of the library.

```
Vector(2.5, 0, 0) // a constant field which is the same for all the agents
```

This vector must then be appropriately mapped the right electrical stimulus for the underlying engine platform of the mobile robot of interest. On top of that, this module exposes several blocks to explore an environment. Particularly, the `brownian` block produces a random velocity vector for each evaluation of the program. In addition to that simple logic, there are movements based on GPS like `goTo` (produces a velocity vector that eventually moves the system to sink at one single point) and `explore` (produces a velocity vector that let the system explore a rectangle defined through `minBound` and `maxBound`). The last one is based on temporal blocks, like `maintainTrajectory` and `maintainUntil`. The former allows the systems to maintain a certain velocity for the time specified. At that moment, a new velocity is generated according to the given strategy. The latter, instead, is used to maintain a certain velocity until a condition is met (e.g., a target position is reached). This module also exposes an `obstacleAvoidance` block (Figure 2d), which creates a vector pointing away from obstacles.

Even if these blocks are quite simple, it is still possible to combine them to create interesting behaviours. For instance, program

```
(maintainVelocity(browian()) + obstacleAvoidance(sense("obs"))).normalize
```

expresses a collective behaviour in which the nodes will explore the environment, while avoiding any obstacles perceived through a sensor. Notice how the composition is achieved by simply summing the computational fields produced by the sub-blocks. Expression `v.normalize` yields `v` as a unit vector (of length 1), while keeping the same direction—useful when combining several vectors together. A summary of the blocks exposed by this module is reported in the following listing:

```
// Movement library
def brownian(scale: Double): Vector
```

```
// GPS Based
def goTo(target: Point3D): Vector
def explore(minBound: Point3D, maxBound: Point3D): Vector
// Temporal Based
def maintainTrajectory(trajectory: => Vector)(time: FiniteDuration):Vector
def maintainUntil(direction: Vector)(condition: Boolean): Vector
// Obstacle Avoidance
def obstacleAvoidance(obstacles: List[Vector]): Vector
```

Flocking blocks. In a swarm-like system, it is often necessary to coordinate the movement of the entire swarm, rather than just individual agents, to achieve emergent behaviours, and ensure that the nodes move cohesively, avoid collisions, and strive to be aligned in a common direction. Therefore, in this module, we have implemented the main blocks to support the *flocking* of agents. Several models are available in the literature for this purpose. Particularly, MacroSwarm exposes the Vicsek [48], Cucker-Smale [20], and Reynolds (Figure 2e) [42] models. We have also exposed the individual blocks to implement Reynolds, which are `cohesion`, `separation`, and `alignment`. These blocks can be used individually by higher-level blocks to implement specific behaviours (e.g., following a leader while avoiding collisions).

Another essential aspect that emerges at this level is the concept of a *variable neighbourhood*. Indeed, it may happen that the logical neighbourhood model used by aggregate computing does not match the one used to coordinate the agents. Thus, the node's visibility can be more *restrictive* or *extensive* according to the neighbourhood model applied. In particular, in the case of Reynolds, it is typical for the separation range to be different from that of alignment. Therefore, the flocking blocks accept a "query" strategy towards a variable neighbourhood. The main implementation of these queries are:

- `OneHopNeighborhood`: the same as the aggregate computing model;
- `OneHopNeighborhoodWithinRange(radius: Double)`: it takes all the nodes in the neighbourhood within the given range.

The flocking models are typically described by an iterated function in which the velocity at time $t+1$ depends on the velocity at time t. Taking as an example the Vicsek rule, it is described as: $v_i(t+1) = \frac{\sum_{j \in \mathcal{N}} v_j(t)}{|\mathcal{N}|} + \eta_i(t)$ where $\mathcal{N}$ is the neighbourhood of the node i at time t, $v_i(t)$ is the velocity of the node i at time t, and $\eta_i(t)$ is a random vector that models the noise of the model. For this reason, each block receives the previous velocity field as a parameter, rather than encoding it internally within each block. This is because the previous velocities may be influenced by other factors, such as constant movements or a target position. Typical usage of this operator follows the following schema:

```
rep(initialVelocity) { oldVelocity => flockingOperator(oldVelocity, ..) }
```

For example, the following program describes a collective movement in which the nodes try to reach the position `(x,y)` while maintaining a distance of `k` meters from one another:

```
rep(Point2D.Zero) {
  v => (goTo(Point2D(x, y)) +
       separation(v, OneHopNeighbourhoodWithinRange(k))).normalize
}
```

Leader-based blocks. These blocks allow agents to follow a designated leader. The idea behind leadership in swarm systems is that a leader can act as a coordinator, influencing the followers that recognise it as such. In the context of aggregate computing, leaders are typically defined as Boolean fields holding `true` for leaders and `false` for non-leaders. Leaders can be predetermined (i.e., nodes with certain characteristics), virtual (i.e., nodes that do not actually exist in the system but are simulated for collective movement steering), or chosen in space (e.g., using the `S` block—see Section 3). A leader can be thought of as creating an *area of influence*, affecting the actions of its followers.

Currently, we have identified `alignWithLeader` and `sinkAt` (Figure 2b) as essential blocks. The former propagates the leader's velocity throughout its area of influence (e.g., via `G`—see Section 3), with followers adjusting their velocity to it. However, sometimes it may also be desirable to create a sort of attraction towards the leader, so that the nodes remain cohesive with it. For this reason, the `sinkAt` block creates a computational field in which nodes tend to move towards the leader. These blocks are useful for higher-level blocks, such as those associated with the creation of teams or spatial formations.

Team formation blocks. These blocks allow agents to form *teams* or subgroups within the swarm, useful e.g. for work division or situations requiring intervention by few agents. In general, the formation of a team creates a "split" in the swarm logic, conceptually creating multiple swarms with potentially different goals (cf. Figure 2c). One way to create teams is by using the `branch` construct (see Section 3). For example, the following program,

```
def alignVelocity(id: Int) =
  alighWithLeader(id == mid(), rep(browian())(x => x)
branch(mid() < 50) { alignVelocity(0) } { alignVelocity(50) }
```

creates two groups, each of which follows a certain velocity dictated by the leaders (0 and 50).

Other times, one needs to create teams based on the spatial structure of the network or when certain conditions are met. The `teamFormation` block supports this scenario. By internally using `S`, it allows for the creation of teams based on certain spatial constraints expressed through parameters `intraDistance` (i.e., the distance between team members) and `targetExtraDistance` (i.e., the size of the leader's area of influence). It is also possible to create teams based on

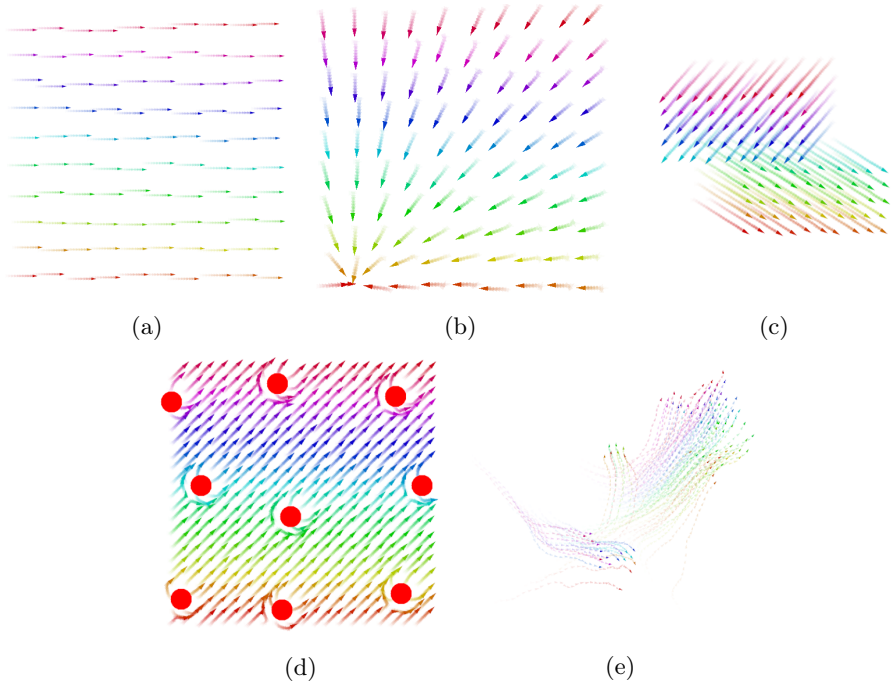


Fig. 2: Overview of swarm behaviours expressible with MacroSwarm.

predetermined leaders, denoted explicitly by Boolean fields. Moreover, since team formation may take time to complete, or require conditions to be met (e.g., that at least N members are present, or that the minimum distance between all nodes is less than a certain threshold), we also parameterise `teamFormation` by a `condition` predicate. An example of built-in predicate is `isTeamFormed`, which verifies that each node under the influence of the leader has a `necessary` a number of neighbours within a `targetDistance` radius. An example is as follows.

```
teamFormation(targetIntraDistance = 30, // separation
  targetExtraDistance = 300, // influence of the leader
  condition = leader => isTeamFormed(leader, targetDistance = 40)
).velocity // use the velocity vector to create the Team
```

Each team must refer to a single leader, who can coordinate the associated nodes (using the APIs exposed by the **Leader Based Block**). In particular, to execute a certain behaviour within a team, the `insideTeam` method must be used. Given the ID of the leader to which a node belongs, this method can define the movement logic relative to that leader. For instance, this code aligns the followers with a velocity generated by a leader,

```
team.insideTeam{leader => alignWithLeader(leader)(rep(brownian())(x => x))}
```

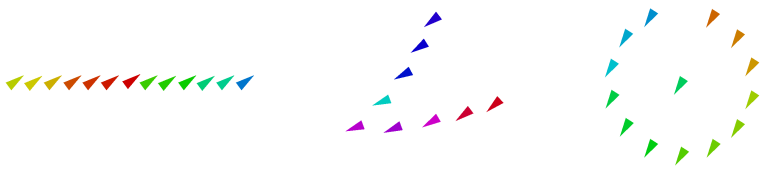


Fig. 3: Examples of the supported patterns. From left to right: line formation, v-like formation, and circular formation.

Pattern formation blocks. Team formation blocks can be used to create groups of agents with certain characteristics. However, sometimes we are also interested in the *spatial structure* of the group. In swarm behaviours, the spatial structures of the teams can be instrumental for performing certain tasks (e.g., coverage or transportation tasks). In MacroSwarm some of the most idiomatic spatial structures are available.

The implementation is as follows. First of all, the formation of structures is based on the presence of a leader that collects the hop-by-hop distances of their followers (leveraging `G` and `C`) and sends them a direction in which they should go to form the required structure (using `G`).

The structures currently supported (Figure 3) are v-like shapes (`vShape`), lines (`line`), and circular formations (`centeredCircle`). These structures are *self-healing*: if there is a disturbance of the structure, the group tends to reconstruct itself and return to a stable structure. Additionally, it is assumed that the leader has his own speed logic. In this way, the group will follow the leader maintaining the chosen structure.

Swarm Planning blocks. With the previous blocks available, there is a need for a handy mechanism to express a series of *plans* that change over time and move the swarm towards different targets. For this reason, MacroSwarm also exposes the concept of *swarm planning*. The idea is to express a series of plans (or missions) defined by a *behaviour* (i.e., the logic of production of a velocity vector) and a *goal* (defined as a boolean predicate condition). At any given time, the swarm will be executing a certain sub-plan, which will be considered complete only when the boolean condition is satisfied. At this point, the swarm will follow the next objective described by the overall plan. The exposed API allows for the creation of these collective plans in the following way:

```
execute.once {
  plan(goTo(goalOne).endWhen(isClose(goalOne)),
  plan(goTo(goalTwo).endWhen(isClose(goalTwo)),
}.run() // will trigger the execution of the plan
```

This snippet creates a plan in which the nodes will first go to `goalOne`, and once reached (`isClose` verifies that the node is close enough to the point passed), it will move on to the next objective `goalTwo`. Since it is specified that the

mission is executed `once`, after the completion of the last plan, the group will stop moving. To make the group repeat the plan, the `repeat` method can be used instead of `once`. Note that there is no coordination between agents in the above code, but you can enforce it using lower-level blocks (e.g., flocking or team-based behaviours). For example, MacroSwarm enables describing a swarm behaviour where: (i) a group of nodes gather around a leader, (ii) the leader brings the entire group towards the `goalOne`, (iii) the leader brings the entire group towards the `goalTwo`. This can be described using the following code:

```
execute.once( // if it is repeated, you can use 'repeat'
  plan{sinkAt(leaderX)}.endWhen{isTeamFormed(leaderX, targetDistance=100)},
  plan(goTo(goalOne)).endWhen{ G(leaderX, isClose(goalOne), x => x)},
  plan(goTo(goalTwo)).endWhen{ G(leaderX, isClose(goalTwo), x => x)},
).run()
```

The use of `G` in this way is a recurrent pattern, and in ScaFi it is exposed through the `broadcast[T](center: Boolean, value: T): T` block.

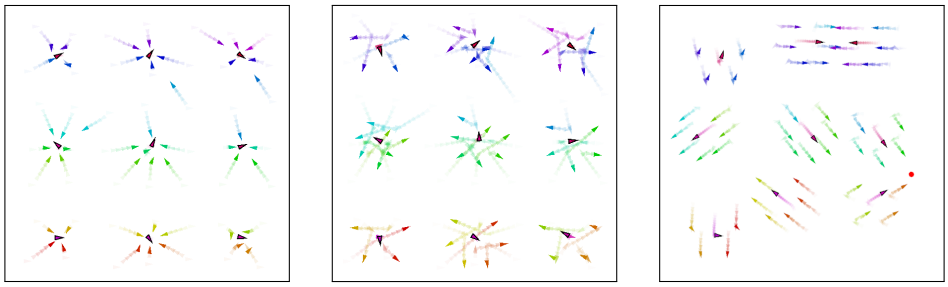
5 Evaluation

To validate the proposed approach and API we define a simulated *find-and-rescue* case study, to show the ability of MacroSwarm to express complex swarm behaviours (Section 5.1). Then, we discuss the results of the case study and the applicability of the proposed approach in real-world scenarios (Section 5.2).

5.1 Case Study: Find and Rescue

In our scenario, we want a fleet of drones to patrol a spatial area. In the area, dangerous situations may arise (e.g., a fire breaks out, a person gets injured, etc.). In response to these, a drone designated as a *healer* must approach and resolve them. Exploration must be carried out in groups composed of *at least* one healer and several *explorers*, who will help the healer identify alarm situations.

Goal. The goal of the proposed case study is to demonstrate the effectiveness of the proposed API in terms of expressiveness (i.e., the ability to describe complex behaviours easily) and correctness (i.e., the described behaviour collectively does what is expressed). For the first point, since it is a qualitative metric, we will show the development process that led to the implementation of the produced code, demonstrating its ease of understanding. For the second point, since deploying a swarm of drones is costly, we will make use of simulations to verify that the program is functioning correctly both qualitatively (e.g., observing the graphical simulation) and quantitatively (i.e., extracting the necessary data and computing metrics that allow us to understand if the system behaves as it should).



(a) Team formation (b) Circle formation (c) Explore

Fig. 4: The first phases of the scenario described in Section 5. At the beginning, the system is split into teams; afterwards, the teams assume a spatial formation (circular, in this case); finally, the teams start exploring the overall area.

Setup. Initially, 50 explorers and 5 healers are randomly positioned in an area of $1km^2$. Each drone has a maximum speed of approximately 20 km/h and a communication range of 100 meters. The alarm situations are randomly generated at different times within the spatial area in a $[0, 50]$ minutes time-frame. Each simulation run lasts 90 minutes, during which we expect the number of alarm situations to reach a minimum value. The node should form teams of at least one healer and several explorers, maintaining a distance of at least 50 meters between the node and the leader

Implementation details. To structure the desired swarm behaviour, we break the problem into parts:

1. the swarm must split into teams regulated by a healer, who works as a *leader* (Figure 4a);
2. teams must assume a spatial formation promoting the efficiency of the exploration (Figure 4b);
3. the teams must explore the overall area (Figure 4c);
4. when any node detects an alarm zone, it must point that to the healer;
5. the healer node approaches the dangerous situation to fix it;
6. then, the team should return to the exploration phase.

We now describe the implementation of each part, leveraging the MacroSwarm API. First of all, for creating teams, we can use the **Team Formation** blocks:

```
val teamFormedLogic =
  (leader: ID) => isTeamFormed(leader, minimumDistance + confidence)
def createTeam() =
  teamFormation(sense("healer"), minimumDistance, teamFormedLogic)
```

where `minimumDistance` is the minimum distance between nodes during the team formation phases and `confidence` is the confidence interval used to check

if the team is formed through the `isTeamFormed` method. Each team then should follow the aforementioned steps, expressible using the **Swarm Planning** API:

```
def insideTeamPlanning(team: Team): Vector =
 team.insideTeam {
  healerId =>
   val leading = healerId == mid() // team leader
   execute.repeat(
    plan(formation(leading)).endWhen(circleIsFormed), // shape formation
    plan(wanderInFormation(leading)).endWhen(dangerFound), // exploration
    plan(goToHealInFormation(leading, inDanger)).endWhen(dangerReached),
    plan(heal(healerId, inDanger)).endWhen(healed(dangerFound)) // healing
   ).run() // repeat the plan
 }
```

The first step is the formation of the teams, based on method `formation` which internally uses `centeredCircle` to place the nodes in a circle around the leader node. Function `circleIsFormed` verifies whether the nodes are in a circle formation, i.e., that the distance between any node and the leader is less than `radius` (set to 50 meters in this scenario). The second step is the exploration phase, implemented by method `wanderInFormation`, which uses the `explore` function to move the nodes to a random direction within given bounds while keeping the circle formation. This leverages `centeredCircle`, passing the movement logic of the healer (leader) to the block. Exploration will go on until someone finds a danger node, denoted by predicate `dangerFound`. This internally uses `C` and `G` to collect the danger nodes' psitions and share them within the team:

```
def dangerFound(healer: Boolean): Boolean = {
  val dangerNodes =
    C(sense("healer"), combinePosition, List(sense("danger")), List.empty)
  broadcast(healer, dangerNodes.nonEmpty)
}
```

The third step is the movement towards the danger node, which is implemented by the `goToHealInFormation` method, which uses again the `centeredCircle` function with a delta vector that moves the leader node towards the danger node. `inDanger` is computed similarly to `dangerFound`, but, in this case, the position will be shared instead. `dangerReached` is a Boolean field indicating if the healer node is close enough to the danger node. The last step is the healing of the danger node, which is modelled as an actuation of the healer. The rescue ends when the danger node is healed. As a final note, we also want the nodes to be able to avoid each other when they are too close, even if they are not in the same team. For this, we leverage the **Flocking** API the `separation` block outside the team logic. Then, the main program is as follows:

```
val team = createTeam()
rep(Vector.Zero) { v =>
  insideTeamPlanning(team) +
  separation(v, OneHopNeighbourhoodWithinRange(avoidDistance))
```

```
}.normalize
```

This program shows that the API is flexible enough to create complex behaviours handling various coordination aspects.

Results. We validated the results by effectively running simulations, publicly available at `https://zenodo.org/badge/latestdoi/611692727`. For this task, we used Alchemist [40], a general simulator for multi-agent and pervasive systems. We launched 64 simulation runs with different random seeds: Figure 5 shows the average results obtained. We extracted the following data:

- *intra-team distance*: after an initial adjustment phase, the system should converge to an average distance of 50 meters (Figure 5a);
- *minimum distance between each node*: as we want to avoid collisions, the minimum distance between two nodes should always be greater than zero (Figure 5b);
- *number of nodes in danger*: we expect the nodes in danger to increase up to 50 minutes and then decrease, tending towards zero (Figure 5c).

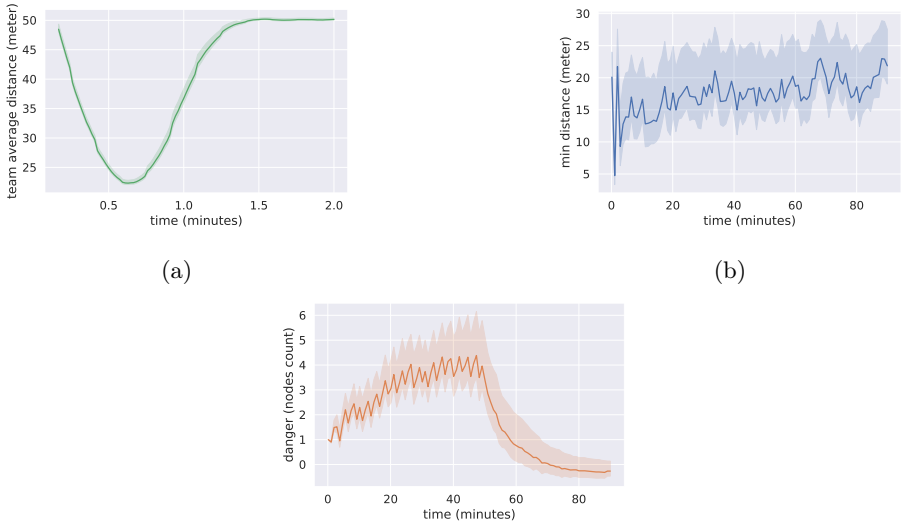
The results (Figure 5) show that the system can achieve the expected outcomes.

5.2 Discussion

Despite its simplicity, this use case allowed us to demonstrate the capability of MacroSwarm, both in qualitative terms (i.e., the produced code is simple and understandable) and quantitative terms (i.e., the data show that the swarm follows the given instructions correctly).

That being said, there are several things to consider when using the library in real-world contexts. Ours is a top-down approach, in which we have defined an evaluation and implementation system that is general enough to be executed in various multi-robot systems. Specifically, we require that at least: *i)* nodes can perceive and interact with neighbours and approximate a direction vector to each of them; *ii)* they can move in a specific direction with a certain velocity; and *iii)* they can perceive distance and direction for certain obstacles. As for point *i)*, this can be developed using specific local sensors (e.g., range and bearing systems [11]), by using GPS, by approximating distances using cameras mounted on each drone, or by using Bluetooth direction finding [43]. Concerning the point *ii)* the velocity vector can be mapped to the motors of the UAVs, or the motor's wheels of the ground robots [27], so it can be easily implemented in real case scenarios. Finally, concerning *iii)*, there are several solutions for perceiving the direction of obstacles by leveraging various sensors, like *Laser Imaging Detection And Ranging (LIDAR)* systems [37].

That being said, we know that the reality gap for real-world scenarios could introduce divergences from the behaviours shown, as the used simulator, although general, does not simulate many aspects of reality, such as communication delay, friction, and possible perception errors. We aim to test the API in more realistic simulators (like Gazebo [26]) or real systems as a future work.



(a) (b)

(c)

Fig. 5: Quantitative plots of the simulated scenario. Figure 5a shows the average team distance in the first two minutes. Figure 5b shows the minimum distance between nodes. Figure 5c shows the nodes in danger through time. Since we run several simulations, the lines show the average values, whereas the area around the lines shows the confidence interval throughout the simulations.

6 Related Work

Related programming approaches for swarms include Meld [4], Buzz [41], Voltron [34], TeCoLa [29], Dolphin [30], Maple-Swarm [28], PARoS [22], Resh [15], and [51]. In the following, we review the works that are more related to MacroSwarm, which are those for expressing *decentralised* behaviours.

Buzz [41] is a mixed imperative-functional language for programming swarms. In Buzz, swarms are first-class abstractions: they can be explicitly created, manipulated, joined (e.g., based on local conditions), and used as a way to address individual members (e.g., for tasking them). For individual robots, the language provides access to local features and the local set of neighbours, for interaction. For swarm-wide consensus, a notion of *virtual stigmergy* is leveraged, based on distributed tuple spaces. Buzz is designed to be an extensible language, since new primitives can be added. Indeed, Buzz is based on a set of quite effective but ad-hoc mechanisms. By contrast, MacroSwarm uses few general and expressive primitives, and supports swarm programming through a library of reusable, composable blocks. Additionally, MacroSwarm can leverage theoretical results from field calculi [50,49], making programs amenable for formal analysis.

Voltron [34] is a programming model for team-level design of drone systems. It represents a group of individual drones through a *team abstraction*, which is

responsible for the overall task. The details of individual drone actions and their timing are delegated to the platform system during runtime. The programmer issues *action commands* to the drone team, along with *spatiotemporal constraints*. The tasks in Voltron are associated with spatial locations, and the team self-organises to populate *multisets of future values* that represent the task's eventual result at a specific location. However, Voltron is imperative in nature, limiting the compositionality of team-level behaviours.

Meld [4] is a logic-based language for programming modular ensembles, for systems where communication is limited to immediate neighbours. It leverages *facts with side-effects* to handle actuation, *production rules* to generate new facts from existing facts, and *aggregate rules* to combine multiple facts into one fact by folding (e.g., maximisation or summation). The runtime deals with communication of facts and removal of invalidated facts. The declarativity and logical foundation make Meld an interesting macroprogramming system; however, it is not clear how it can scale with the complexity of general swarm behaviour. Indeed, it is mainly adopted for shape formation and self-reconfiguring ensembles.

Finally, we mention another category of related works, which are *task orchestration languages* for swarms (e.g., TeCoLa [29], Dolphin [30], Maple-Swarm [28], PARoS [22], Resh [15], and [51]): they adopt quite a different approach that leverages centralised entities to control the activity of the swarm members based on the provided task descriptions.

7 Conclusion and Future Work

We presented MacroSwarm, a framework for top-down swarm programming that provides composable blocks capturing common decentralised swarm behaviours. It builds on aggregate computing, a formally-grounded field-based coordination paradigm, and is implemented on top of the ScaFi toolkit/DSL. We show through examples and a simulated case study that the approach is compositional, practical, and expressive.

As future work, we plan to make the API more comprehensive, by covering all the main patterns from notable taxonomies of swarm behaviour [13]. Additionally, it would be interesting to investigate approaches for synthesising compositions of MacroSwarm blocks, e.g., by following the reinforcement learning-based approach of [3]. Last but not least, we would like to deploy and test the framework on real testbeds.

Acknowledgements. This work was supported by the Italian PRIN project "CommonWears" (2020HCWWLP) and the EU/MUR FSE PON-R&I 2014-2020.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming interactions in collective adaptive systems by relying on attribute-based communication. Science of Computer Programming **192** (2020)

2. Aguzzi, G., Casadei, R., Maltoni, N., Pianini, D., Viroli, M.: Scafi-web: A web-based application for field-based coordination programming. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 285–299. Springer (2021), https://doi.org/10.1007/978-3-030-78142-2_18
3. Aguzzi, G., Casadei, R., Viroli, M.: Towards reinforcement learning-based aggregate computing. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13271, pp. 72–91. Springer (2022), https://doi.org/10.1007/978-3-031-08143-9_5
4. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 2794–2800. IEEE (2007), https://doi.org/10.1109/IROS.2007.4399480
5. Audrito, G., Beal, J., Damiani, F., Viroli, M.: Space-time universality of field calculus. In: Serugendo, G.D.M., Loreti, M. (eds.) Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018. Proceedings. Lecture Notes in Computer Science, vol. 10852, pp. 1–20. Springer (2018), https://doi.org/10.1007/978-3-319-92408-3_1
6. Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M.: Optimal resilient distributed data collection in mobile edge environments. Comput. Electr. Eng. **96**(Part), 107580 (2021), https://doi.org/10.1016/j.compeleceng.2021.107580
7. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: Functional programming for distributed systems with XC. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany. LIPIcs, vol. 222, pp. 20:1–20:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022), https://doi.org/10.4230/LIPIcs.ECOOP.2022.20
8. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Computation Against a Neighbour: Addressing Large-Scale Distribution and Adaptivity with Functional Programming and Scala. Logical Methods in Computer Science **Volume 19, Issue 1** (Jan 2023), https://lmcs.episciences.org/10826
9. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic **20**(1), 5:1–5:55 (Jan 2019), http://doi.acm.org/10.1145/3285956
10. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013). https://doi.org/10.4018/978-1-4666-2092-6.ch016
11. Bilaloglu, C., Sahin, M., Arvin, F., Sahin, E., Turgut, A.E.: A novel time-of-flight range and bearing sensor system for micro air vehicle swarms. In: Dorigo, M., Hamann, H., López-Ibáñez, M., García-Nieto, J., Engelbrecht, A.P., Pin-

ciroli, C., Strobel, V., Camacho-Villalón, C.L. (eds.) Swarm Intelligence - 13th International Conference, ANTS 2022, Málaga, Spain, November 2-4, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13491, pp. 248–256. Springer (2022). https://doi.org/10.1007/978-3-031-20176-9_20, https://doi.org/10.1007/978-3-031-20176-9_20

12. Bonabeau, E., Dorigo, M., Théraulaz, G.: Swarm intelligence: from natural to artificial systems. Santa Fe Institute Studies in the Sciences of Complexity, Oxford university press (1999)
13. Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. **7**(1), 1–41 (2013), https://doi.org/10.1007/s11721-012-0075-2
14. Busoniu, L., Babuska, R., Schutter, B.D.: A comprehensive survey of multiagent reinforcement learning. IEEE Trans. Syst. Man Cybern. Part C **38**(2), 156–172 (2008), https://doi.org/10.1109/TSMCC.2007.913919
15. Carroll, M., Namjoshi, K.S., Segall, I.: The Resh programming language for multi-robot orchestration. In: IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021. pp. 4026–4032. IEEE (2021), https://doi.org/10.1109/ICRA48506.2021.9561133
16. Casadei, R.: Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Computing Surveys (Jan 2023), https://doi.org/10.1145/3579353
17. 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 **12**(11), 203 (2020), https://doi.org/10.3390/fi12110203
18. Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: ScaFi: A Scala DSL and toolkit for aggregate programming. SoftwareX **20**, 101248 (2022), https://doi.org/10.1016/j.softx.2022.101248
19. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artif. Intell. **97**, 104081 (2021), https://doi.org/10.1016/j.engappai.2020.104081
20. Cucker, F., Smale, S.: Emergent behavior in flocks. IEEE Transactions on Automatic Control **52**(5), 852–862 (2007). https://doi.org/10.1109/TAC.2007.895842
21. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014)
22. Dedousis, D., Kalogeraki, V.: A framework for programming a swarm of UAVs. In: 11th PErvasive Technologies Related to Assistive Environments Conference (PETRA'18), Proceedings. pp. 5–12. ACM (2018), https://doi.org/10.1145/3197768.3197772
23. Dorigo, M., Theraulaz, G., Trianni, V.: Reflections on the future of swarm robotics. Sci. Robotics **5**(49), 4385 (2020), https://doi.org/10.1126/scirobotics.abe4385
24. Galinina, O., Mikhaylov, K., Huang, K., Andreev, S., Koucheryavy, Y.: Wirelessly powered urban crowd sensing over wearables: Trading energy for data. IEEE Wirel. Commun. **25**(2), 140–149 (2018), https://doi.org/10.1109/MWC.2018.1600468
25. Gershenson, C., Trianni, V., Werfel, J., Sayama, H.: Self-organization and artificial life. Artif. Life **26**(3), 391–408 (2020), https://doi.org/10.1162/artl_a_00324
26. Koenig, N.P., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent

Robots and Systems, Sendai, Japan, September 28 - October 2, 2004. pp. 2149–2154. IEEE (2004). https://doi.org/10.1109/IROS.2004.1389727
27. Koren, Y., Borenstein, J.: Potential field methods and their inherent limitations for mobile robot navigation. In: Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, USA, 9-11 April 1991. pp. 1398–1404. IEEE Computer Society (1991). https://doi.org/10.1109/ROBOT.1991.131810, https://doi.org/10.1109/ROBOT.1991.131810
28. Kosak, O., Huhn, L., Bohn, F., Wanninger, C., Hoffmann, A., Reif, W.: Maple-swarm: Programming collective behavior for ensembles by extending HTN-planning. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 507–524. Springer (2020), https://doi.org/10.1007/978-3-030-61470-6_30
29. 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). pp. 115–124. ACM (2016), https://doi.org/10.1145/2994374.2994397
30. Lima, K., Marques, E.R.B., Pinto, J., Sousa, J.B.: Dolphin: A task orchestration language for autonomous vehicle networks. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2018, Madrid, Spain, October 1-5, 2018. pp. 603–610. IEEE (2018), https://doi.org/10.1109/IROS.2018.8594059
31. Lluch-Lafuente, A., Loreti, M., Montanari, U.: Asynchronous distributed execution of fixpoint-based computational fields. Log. Methods Comput. Sci. **13**(1) (2017)
32. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. ACM Comput. Surv. **52**(5), 100:1–100:41 (2019), https://doi.org/10.1145/3342355
33. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: Pervasive Computing and Communications, 2004. pp. 263– 273. IEEE (2004)
34. Mottola, L., Moretta, M., Whitehouse, K., Ghezzi, C.: Team-level programming of drone sensor networks. In: Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys'14). pp. 177–190. ACM (2014), https://doi.org/10.1145/2668332.2668353
35. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87 (2004)
36. Nicola, R.D., Jähnichen, S., Wirsing, M.: Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. **22**(4), 389–397 (2020), https://doi.org/10.1007/s10009-020-00565-0
37. Peng, Y., Qu, D., Zhong, Y., Xie, S., Luo, J., Gu, J.: The obstacle detection and obstacle avoidance algorithm based on 2-d lidar. In: IEEE International Conference on Information and Automation, ICIA 2015, Lijiang, China, August 8-10, 2015. pp. 1648–1653. IEEE (2015). https://doi.org/10.1109/ICInfA.2015.7279550, https://doi.org/10.1109/ICInfA.2015.7279550
38. Pianini, D., Casadei, R., Viroli, M.: Self-stabilising priority-based multi-leader election and network partitioning. In: Casadei, R., Nitto, E.D., Gerostathopoulos, I., Pianini, D., Dusparic, I., Wood, T., Nelson, P.R., Pournaras, E., Bencomo, N.,

Götz, S., Krupitzer, C., Raibulet, C. (eds.) IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022, Virtual, CA, USA, September 19-23, 2022. pp. 81–90. IEEE (2022), https://doi.org/10.1109/ACSOS55765.2022.00026

39. Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F.: Time-fluid field-based coordination through programmable distributed schedulers. Log. Methods Comput. Sci. **17**(4) (2021), https://doi.org/10.46298/lmcs-17(4:13)2021
40. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation **7**(3), 202–215 (2013), https://doi.org/10.1057/jos.2012.27
41. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2016, Daejeon, South Korea, October 9-14, 2016. pp. 3794–3800. IEEE (2016), https://doi.org/10.1109/IROS.2016.7759558
42. Reynolds, C.W.: Flocks, herds and schools: A distributed behavioral model. In: Stone, M.C. (ed.) Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987, Anaheim, California, USA, July 27-31, 1987. pp. 25–34. ACM (1987), https://doi.org/10.1145/37401.37406
43. Sambu, P., Won, M.: An experimental study on direction finding of bluetooth 5.1: Indoor vs outdoor. In: IEEE Wireless Communications and Networking Conference, WCNC 2022, Austin, TX, USA, April 10-13, 2022. pp. 1934–1939. IEEE (2022). https://doi.org/10.1109/WCNC51071.2022.9771930, https://doi.org/10.1109/WCNC51071.2022.9771930
44. Schranz, M., Umlauft, M., Sende, M., Elmenreich, W.: Swarm robotic behaviors and current applications. Frontiers Robotics AI **7**, 36 (2020), https://doi.org/10.3389/frobt.2020.00036
45. Tahir, A., Böling, J., Haghbayan, M.H., Toivonen, H.T., Plosila, J.: Swarms of unmanned aerial vehicles - A survey. J. Ind. Inf. Integr. **16**, 100106 (2019), https://doi.org/10.1016/j.jii.2019.100106
46. Trianni, V.: Evolutionary Swarm Robotics - Evolving Self-Organising Behaviours in Groups of Autonomous Robots, Studies in Computational Intelligence, vol. 108. Springer (2008), https://doi.org/10.1007/978-3-540-77612-3
47. Varenne, F., Chaigneau, P., Petitot, J., Doursat, R.: Programming the emergence in morphogenetically architected complex systems. Acta biotheoretica **63**(3), 295–308 (2015). https://doi.org/10.1007/s10441-015-9262-z
48. Vicsek, T., Czirók, A., Ben-Jacob, E., Cohen, I., Shochet, O.: Novel type of phase transition in a system of self-driven particles. Phys. Rev. Lett. **75**, 1226–1229 (Aug 1995), https://link.aps.org/doi/10.1103/PhysRevLett.75.1226
49. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 16:1–16:28 (2018), https://doi.org/10.1145/3177774
50. 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. **109** (2019)
51. Yi, W., Di, B., Li, R., Dai, H., Yi, X., Wang, Y., Yang, X.: An actor-based programming framework for swarm robotic systems. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021. pp. 8012–8019. IEEE (2020), https://doi.org/10.1109/IROS45743.2020.9341198